TECHNICAL SPECIFICATION

Midgard L2: Scaling Cardano with Optimistic Rollups

WORKING DRAFT
2025-10-22

FluidTokens ANASTASIA LABS

Midgard L2: Scaling Cardano with Optimistic Rollups

Philip DiSarro
Anastasia Labs

Jonathan Rodriguez
Anastasia Labs
info@anastasialabs.com

George Flerovsky

Abstract

Midgard is the first optimistic rollup protocol on Cardano. It offloads transaction processing from Layer 1 (L1) to Layer 2 (L2) while maintaining L1's security and decentralization properties for those transactions. As a result, it can handle a significantly higher volume of transactions without compromising on security.

This whitepaper outlines the interactions between L1 and L2, the role of operators and watchers, and the processes for managing state transitions and resolving disputes. The optimistic rollup model implemented in Midgard represents a significant advancement in the scalability and security of general-purpose Layer 2 solutions on Cardano.

We discuss the economic incentives that ensure the protocol's integrity and efficiency. Operators guarantee the validity of their committed blocks by posting bonds, which are slashable if the blocks are proven invalid. Members of the public are incentivized to watch the operators' blocks, and they are rewarded with a portion of the slashed bonds when their fraud proofs prevent invalid blocks from merging into Midgard's confirmed state.

Midgard's architecture leverages a family of smart contracts on Cardano's L1 to manage state transitions and enforce the security of the L2 operations. Key components include a robust operator management system, efficient storage for transaction blocks committed from the L2, and a comprehensive mechanism for submitting and validating fraud proofs. Overall, Midgard's design aims to provide a scalable and secure solution for the growing needs of the Cardano ecosystem.

Contributors

Raul Antonio
Fluid Tokens

Matteo Coppola
Fluid Tokens

fallen-icarus
P2P-Defi

Riley Kilgore
IOG

Keyan Maskoot
Anastasia Labs

Bora Oben
Anastasia Labs

Mark Petruska
Anastasia Labs

Kasey White
Cardano Foundation

Branch: (None) @ 8799ca9 • Commit date: 2025-10-22

Contents

Introduction

Midgard is a Layer 2 (L2) scaling solution for the Cardano blockchain. It employs optimistic rollup technology to enhance Cardano's capacity to process transactions and host more complex applications, delivering a richer user experience at a more competitive cost. As Cardano continues to grow in usage and demand, scaling solutions like Midgard are critical for maintaining high performance and low transaction costs. This whitepaper describes the architecture and technical design of the Midgard protocol, detailing how it integrates with Cardano's Layer 1 (L1) to deliver secure and efficient transaction processing.

Optimistic rollups process blocks of transactions off-chain and commit those blocks' headers onchain to the L1 ledger. Each block is committed by a Midgard operator who guarantees the block's validity. The block then waits in a queue for at least a fixed duration to be merged into the confirmed state of the optimistic rollup on L1. The operator must collateralize their guarantee with a bond deposit and publish the full contents of the block on the publicly accessible data availability (DA) layer.

While a committed block is queued, anyone can inspect its contents on the data availability layer and ascertain whether it is valid. If someone detects that the block is invalid, they can submit a fraud proof to prevent it from being merged into the confirmed state, slash the operator's bond, and receive a portion of the forfeited bond as a reward. Thus, an optimistic rollup can process a large number of transactions offchain while maintaining security and finality properties that are similar to transactions processed directly onchain, as long as:

- The bond requirement for the rollup's operators is large enough to discourage fraud.
- The reward for preventing an invalid block from merging is large enough to encourage public vigilance in watching the operators.
- The waiting period for committed blocks is long enough to allow the watchers to detect and prove fraud before those blocks are merged.
- The data availability layer is accessible by anyone who wishes to inspect the rollup blocks at any time that they wish to do so.

Whenever the latter three security parameter values are calibrated to provide a high probability of invalid blocks being detected and disqualified, the bond requirement is a strong deterrent against operators attempting fraud. An operator cannot dismiss the forfeited bond as merely a "cost of doing business" paid to obtain potentially larger revenues from fraud. Whenever an invalid block is disqualified, it does *not* affect the confirmed state, so there are no revenues from that fraud to offset the operator's forfeited bond.

The main design goal of Midgard is to streamline the processes by which blocks are committed/merged, fraud is detected, and fraud proofs are verified onchain. Advancing this goal allows the security parameters to be calibrated to achieve a better balance between security, transaction throughput, confirmation time, and community participation in committing blocks and detecting fraud.

Scalability and efficiency

By processing transactions off-chain and only validating them on-chain when fraud proofs challenge them, Midgard significantly increases throughput and reduces costs for Cardano transactions. Its rollup blocks use sparse Merkle trees and compact state representations to enhance the protocol's efficiency further, enabling it to handle a large volume of transactions without overburdening the L1.

The deterministic nature of Cardano transactions allows Midgard fraud proofs to pinpoint the specific site of a transaction that violated Midgard's ledger rules, without having to look at any other parts of that transaction, any other unrelated transactions within the block, or any other blocks. This keeps fraud proofs and their onchain validation procedures small and efficient, which reduces the time and cost needed to submit fraud proofs when invalid blocks are detected, which makes it feasible for a wider group of people to police Midgard's blocks. In this way, Midgard significantly reduces fraud proof size relative to optimistic rollups used in Ethereum and other account-based blockchain ecosystems, where a much larger part of the global blockchain state needs to be inspected when constructing and verifying a fraud proof.

Censorship resistance and fallback mechanisms

On its own, the optimistic rollup mechanism described above ensures a high-level of assurance for the validity of block headers committed to the state queue and merged to Midgard's confirmed state. However, it does not prevent operators from censoring users' deposits, withdrawals, and L2 transactions. Consequently, Midgard's consensus protocol includes additional smart contract mechanisms to provide censorship resistance for these events.

Midgard deposits and withdrawals are initiated via L1 smart contracts that assign definite inclusion times to them. An operator block is invalid if it contains these inclusion times in its event interval but fails to include the associated deposit or withdrawal events. This ensures that if operators continue committing blocks to Midgard's state queue, then they cannot ignore deposit and withdrawal events.

Midgard L2 transaction requests are typically submitted to operators via a publicly accessible API, and they can be ignored by operators. However, any user can escalate his L2 transaction request by posting a transaction order on L1. Similar to Midgard deposits and withdrawals, an L1 transaction order is assigned an inclusion time that guarantees its inclusion in a subsequent valid block.

If Midgard operators stop committing blocks at all to the state queue, then the inclusion times on their own cannot guarantee that deposits, withdrawals, and L2 transactions will be processed in a timely manner. However, for this extreme case, Midgard's consensus protocol includes the escape hatch mechanism, which temporarily lowers the bond requirement for new operators. This makes it easier for honest actors to become Midgard operators and start committing blocks to the state queue. This ensures that user funds cannot be stranded on Midgard even if its current operators entirely stop committing blocks.

Feedback and contributions welcome!

This document specifies the ledger, onchain contracts, and offchain components of the Midgard architecture. We detail the roles of users, operators, and watchers, how blocks are committed and merged on L1, and how fraud proofs are handled.

We invite developers, validators, and the broader community to engage with this document and offer feedback and contributions to help refine and enhance the protocol.

Chapter 1

Ledger state

Midgard's L2 ledger consists of a chain of blocks. Each block defines a transition from the previous block's set of unspent transaction outputs (utxos) to a new utxo set. Unlike Cardano's closed-system L1 ledger, Midgard's open-system L2 ledger allows block transitions to create and destroy utxos in response to exogenous events—namely, deposits and withdrawal requests that occur on the L1 ledger. This is in addition to Midgard's endogenous L2 transactions, which work the same as L1 transactions but with reduced functionality (no staking/governance actions).[1]

The blocks are stored temporarily on Midgard's data availability layer and permanently on Midgard's archive nodes. The blocks' headers are committed to Midgard's L1 state queue data structure to establish immutability for the blocks' sequence and contents as part of Midgard's L1 contract-based consensus protocol. Block headers have a fixed byte size regardless of how many deposit, transaction, and withdrawal events are held by their blocks—this size leverage between blocks and headers is how Midgard multiplies Cardano's transaction throughput.

Midgard L1 contract-based consensus protocol irreversibly considers a block to be confirmed as soon as all its predecessors are confirmed and the block's maturity period has elapsed, as long as no fraud proofs have been verified on L1 to prove that the block violates one of Midgard's ledger rules.

The irreversibility of block confirmation allows the L1 representation of confirmed block headers to be condensed. For instance, it can stop tracking confirmed withdrawal requests after they are paid out and confirmed deposits after they are absorbed into Midgard's reserve. It can avoid tracking any confirmed transactions and only needs to track the last confirmed block's utxo set, as it is required to confirm the next block. Finally, it can drop all other header data for the last confirmed block's predecessors, as it is implicitly tracked by a chained hash in the last confirmed header and is not required to confirm the next block.

As a result, Midgard's L1 confirmed state consists of a fixed-byte-size record with selected fields from the last confirmed block header and a variable-size dataset tracking confirmed withdrawal requests and deposits until they are processed.

[1] Cardano developers concerned about the absence of the "Withdraw 0" action need not worry. Midgard's ledger rules permit the "Observe" script purpose defined in CIP-112, which is a more principled replacement for "Withdraw 0" that is expected to arrive in the next era of Cardano mainnet in 2025.

1.1 Block

A block consists of a header hash, a header, and a block body:

$$\text{Block} := \left\{ \begin{array}{ll} \text{header\_hash} : & \text{HeaderHash} \\ \text{header} : & \text{Header} \\ \text{block\_body} : & \text{BlockBody} \end{array} \right\}$$

The block body contains the block's transactions, deposits, and withdrawals, along with the unspent outputs that result from applying the block's transition to the previous block's unspent outputs. The transactions include both transaction requests (submitted to operators' mempools) and transaction orders (submitted as L1 events).

$$\text{BlockBody} := \left\{ \begin{array}{ll} \text{utxos} : & \text{UtxoSet} \\ \text{transactions} : & \text{TxSet} \\ \text{deposits} : & \text{DepositSet} \\ \text{withdrawals} : & \text{WithdrawalSet} \end{array} \right\}$$

Figure 1.1.1 shows how the block's `utxos` are derived from the previous block's `utxos` by applying the block's events. The events are applied in the following order: withdrawals, then transactions (with transaction orders prioritized over transaction requests), and finally deposits. Failing to uphold proper ordering of event application will be considered fraud. This ensures transaction orders (which are costlier) are not marked invalid in favor of transaction requests.

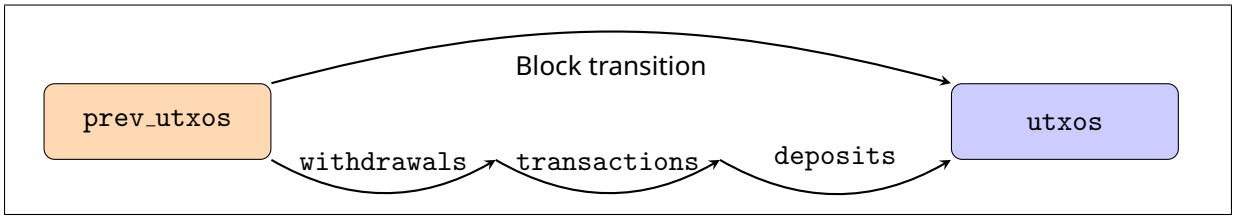

Figure 1.1.1: A block's transition from a previous block's utxo set to a new utxo set.

The block is what gets serialized and stored on Midgard's data availability layer. During serialization, each of the block body's sets is serialized as a sequence of pairs, sorted in ascending order on the unique key of each element.

However, only the header hash and header are stored on Cardano L1. This is sufficient because the header specifies Merkle Patricia Trie (MPT) root hashes for each of the sets in the block body. Each of these root hashes can be verified onchain by streaming over the corresponding set's elements, hashing them, and iteratively calculating the root hash.

Figure 1.1.2 shows an example MPT representation of a block's transactions. The $(\text{TxId}_i, \text{MidgardTx}_i)$ pairs are the data blocks of the tree, which are individually hashed to form the leaves L_i of the tree. The hashes of sibling leaves are concatenated and hashed to form their parent nodes. The nodes N_{ij} are formed by concatenating and hashing their children's hashes. The `transactions_root` is formed by concatenating and hashing its children's hashes.

1.1.1 Transaction requests vs. transaction orders

To distinguish between transactions applied to the ledger from different sources, Midgard recognizes two types of transactions within a block:

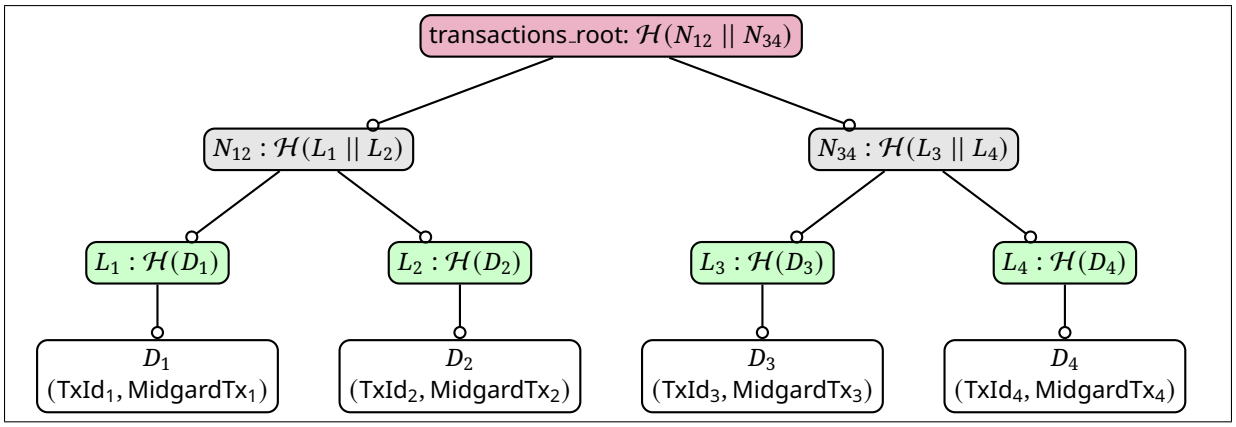

Figure 1.1.2: A Merkle Patricia Trie example for a block's transactions.

- **Transaction requests:** Transactions submitted to operators' mempools. These are the typical L2 transactions that operators collect from users offchain.
- **Transaction orders:** Transactions submitted as L1 events through the transaction order contract (see Section 1.5). These provide an alternative submission path when users need guaranteed inclusion.

Both transaction requests and transaction orders are included in the block's `transactions` set and share the same `transactions_root` Merkle root. However, they must be applied to the ledger with transaction orders prioritized over requests.

1.1.2 Block header

A block header is a record with fixed-size fields: integers, hashes, and fixed-size bytestrings. A block header hash is 28 bytes in size and calculated via the Blake2b-224.

$$\text{HeaderHash} := \mathcal{H}_{\text{Blake2b}-224}(\text{Header})$$

$$\text{Header} := \left\{ \begin{array}{ll} \text{prev\_utxos\_root} : & \text{RootHash} \\ \text{utxos\_root} : & \text{RootHash} \\ \text{transactions\_root} : & \text{RootHash} \\ \text{deposits\_root} : & \text{RootHash} \\ \text{withdrawals\_root} : & \text{RootHash} \\ \text{start\_time} : & \text{PosixTime} \\ \text{end\_time} : & \text{PosixTime} \\ \text{prev\_header\_hash} : & \text{HeaderHash} \\ \text{operator\_vkey} : & \text{VerificationKey} \\ \text{protocol\_version} : & \text{Int} \end{array} \right\}$$

These header fields are interpreted as follows:

- The `*_root` fields are the MPT root hashes of the corresponding sets in the block body.
- The `prev_utxos_root` is a copy of the `utxos_root` from the previous block, included for convenience in the fraud proof verification procedures.
- The `start_time` and `end_time` fields are the block's event interval bounds (see Section 3.1).

- The `prev_header_hash` field is a hash of the previous block header. For the genesis block, this field is set to 28 `0x00` bytes.
- The `operator_vkey` field is the cryptographic verification key for signatures of the operator who committed the block header to the L1 state queue.
- The `protocol_version` is the Midgard protocol version that applies to this block.

1.2 Utxo set

A utxo set is a finite map from output reference to transaction output:

$$\begin{aligned}\mathsf{UtxoSet} &:= \mathsf{Map}(\mathsf{OutputRef}, \mathsf{Output}) \\ &:= \left\{(k_i : \mathsf{OutputRef}, v_i : \mathsf{Output}) \mid \forall i \neq j.\ k_i \neq k_j\right\}\end{aligned}$$

An output reference is a tuple that uniquely identifies an output by a hash of the ledger event that created it (either a transaction or a deposit) and its index among the outputs of that event.

$$\mathsf{OutputRef} := \left\{ \begin{array}{ll} \mathsf{id} : & \mathsf{TxId} \\ \mathsf{index} : & \mathsf{Int} \end{array} \right\}$$

An output is a tuple describing a bundle of tokens, data, and a script that have been placed by a transaction at an address in the ledger:

$$\mathsf{Output} := \left\{ \begin{array}{ll} \mathsf{addr} : & \mathsf{Address} \\ \mathsf{value} : & \mathsf{Value} \\ \mathsf{datum} : & \mathsf{Option}(\mathsf{Data}) \\ \mathsf{script} : & \mathsf{Option}(\mathsf{Script}) \end{array} \right\}$$

Within the context of a Midgard block, the utxo set that we are interested in consists of the outputs created by deposits and transactions but not yet spent by transactions and withdrawals, considering all the deposits, transactions, and withdrawals of this block and all its predecessors. This is the utxo set that we transform into an MPT and place into the block body's `utxo` field.

Midgard requires all L2 datums to be inline.

TODO

Midgard uses a different network ID for L2 utxos' addresses.

TODO

1.3 Deposit event

A deposit set is a finite map from deposit IDs to deposit info:

$$\begin{aligned}\mathsf{DepositSet} &:= \mathsf{Map}(\mathsf{DepositId}, \mathsf{DepositInfo}) \\ &:= \left\{(k_i : \mathsf{DepositId}, v_i : \mathsf{DepositInfo}) \mid \forall i \neq j.\ k_i \neq k_j\right\}\end{aligned}$$

A deposit event in a Midgard block acknowledges that a user has created an L1 utxo at the Midgard L1 deposit address, intending to transfer that utxo's tokens to the L2 ledger.

$$\begin{aligned}\mathsf{DepositEvent} &:= (\mathsf{DepositId}, \mathsf{DepositInfo}) \\ \mathsf{DepositId} &:= \mathsf{OutputRef} \\ \mathsf{DepositInfo} &:= \left\{\begin{array}{ll}\mathsf{l2\_address}: & \mathsf{Address} \\ \mathsf{l2\_datum}: & \mathsf{Option}(\mathsf{Data})\end{array}\right\}\end{aligned}$$

The deposit ID corresponds to one of the inputs spent by the user in the L1 transaction that created the L1 deposit utxo. This identifier is needed to find the L1 deposit utxo, ensure that deposit events are unique, and detect when an operator has fabricated a deposit event without the corresponding deposit utxo existing in the L1 ledger.

Suppose a deposit event is permitted by Midgard's ledger rules to be included in a block. In that case, its effect is to add a new L2 utxo to the block's utxo set containing the value from the L1 deposit utxo at the address (`l2_address`) and with the inline datum (`l2_datum`) specified by the user. The L2 output reference of this new utxo is as follows:

$$\mathsf{l2\_outref}(\mathsf{deposit\_id}) := \left\{\begin{array}{ll}\mathsf{id} & := \mathsf{hash}(\mathsf{deposit\_id}) \\ \mathsf{index} & := 0\end{array}\right\}$$

In other words, the L2 ledger treats the new utxo as if it was created by a notional transaction with `TxId` equal to the hash of the deposit ID.

If the block containing the deposit event is confirmed, the corresponding L1 deposit utxo may be absorbed into the Midgard reserve. Section 2.1 describes the lifecycle of a deposit in further detail, including how the deposit event information is validated.

1.4 Withdrawal event

A withdrawal set is a finite map from withdrawal ID to withdrawal info:

$$\begin{aligned}\mathsf{WithdrawalSet} &:= \mathsf{Map}(\mathsf{WithdrawalId}, \mathsf{WithdrawalInfo}) \\ &:= \left\{(k_i : \mathsf{WithdrawalId}, v_i : \mathsf{WithdrawalInfo}) \mid \forall i \neq j.\ k_i \neq k_j\right\}\end{aligned}$$

A withdrawal event in a Midgard block acknowledges that a user has created an L1 utxo at the Midgard L1 withdrawal address, requesting the transfer of an L2 utxo's tokens to the L1 ledger.

$$\begin{aligned}\mathsf{WithdrawalEvent} &:= (\mathsf{WithdrawalId}, \mathsf{WithdrawalInfo}) \\ \mathsf{WithdrawalId} &:= \mathsf{OutputRef}\end{aligned}$$

$$\text{WithdrawalInfo} := \left\{ \begin{array}{ll} \text{l2\_outref} : & \text{OutputRef} \\ \text{l2\_value} : & \text{Value} \\ \text{l1\_address} : & \text{Address} \\ \text{l1\_datum} : & \text{Option(Data)} \end{array} \right\}$$

The `WithdrawalId` of a withdrawal event corresponds to one of the inputs spent by the user in the L1 transaction that created the L1 withdrawal request utxo (more specifically, it's the hash of its serialized output-reference). This key is needed to identify the L1 withdrawal utxo, ensure that withdrawal events are unique, and detect when an operator has fabricated a withdrawal event without the corresponding withdrawal request existing in the L1 ledger.

Any withdrawal request must first be sent to the outbox contract on L2 in order to provide a few guarantees for the L1 order:

1. Consent of owners
2. Since utxos at outbox contract become unspendable via L2 transactions, their persistence in ledger until the inclusion of their withdrawal events on L1 is guaranteed
3. Quantity of tokens do not exceed the corresponding protocol parameter

With the utxo created at the outbox contract, incentives can be adjusted so that operators will continue the withdrawal order on L1, and therefore relieve users from providing the min ADA themselves.

Inclusion of a withdrawal event in a block leads to the removal of the output at output-reference `l2_outref` from the block's utxo set.

Suppose the block containing the withdrawal event is confirmed. The L1 withdrawal request utxo will be sent to the `payout` contract, after which it can be gradually funded from the reserve. Once fully funded, its utxo can be transfered to `l1_address` with `l1_datum` attached.

Section 2.4 describes the lifecycle of a withdrawal request in further detail.

1.5 Transaction Order event

A transaction order set is a finite map from transaction order ID to a Midgard transaction:

$$\begin{aligned} \text{TxOrderSet} &:= \text{Map}(\text{TxOrderId}, \text{MidgardTx}) \\ &:= \left\{ (k_i : \text{TxOrderId}, v_i : \text{MidgardTx}) \mid \forall i \neq j.\ k_i \neq k_j \right\} \end{aligned}$$

Transaction orders are events on L1 as an alternative to the L2 transaction submission to operators. Similar to a deposit or withdrawal event, a transaction order in a Midgard block acknowledges that a user has created an L1 utxo at the Midgard L1 transaction order address, obligating the operator to include the given L2 transaction. In turn, the operator is free to set `MidgardTxValidity`, if the included transaction is invalid. This invalidity claim itself can then be subject to fraud proofs if the operator had stated incorrectly.

$$\begin{aligned} \text{TxOrderEvent} &:= (\text{TxOrderId}, \text{MidgardTx}) \\ \text{TxOrderId} &:= \text{OutputRef} \end{aligned}$$

The `TxOrderId` of a transaction order event corresponds to one of the inputs spent by the user in the L1 transaction that created the transaction order utxo on L1 (more specifically, it's the hash of its serialized output-reference). This key is needed to identify the L1 transaction order utxo, and ensure that each order is unique.

The effect of an included transaction order on a block's utxo set is identical to any other transaction from block's transaction tree. Meaning it simply should remove all the transaction's inputs from the ledger, and add all its outputs.

Section 2.3 describes the lifecycle of a transaction order request in further detail.

1.6 Transaction

A transaction set in Midgard is a finite map from transaction ID to Midgard L2 transaction, where the ID is also constrained to be the Blake2b-256 hash of the transaction:

$$\begin{aligned}\mathsf{TxSet} &:= \mathsf{Map}(\mathsf{TxId}, \mathsf{MidgardTx}) \\ &:= \left\{(k_i : \mathsf{TxId}, v_i : \mathsf{MidgardTx}) \mid k_i \equiv \mathcal{H}_{\mathsf{Blake2b-256}}(v_i),\ \forall i \neq j.\ k_i \neq k_j\right\}\end{aligned}$$

An L2 transaction in a Midgard block is an endogenous event. Its corresponding utxo set transition is validated purely based on the information contained in the utxo set. This contrasts with deposit and withdrawal events, which create and spend utxos (respectively) based on information observed outside the L2 ledger.

A transaction can only spend a utxo if it satisfies the conditions of the spending validator corresponding to the utxo's payment address, and it can only mint and burn tokens by satisfying the conditions of the corresponding minting policies. Of course, users can inject information into the utxo set via redeemer arguments and output datums set in transactions, but those are still subject to the transaction scripts' conditions.

1.6.1 Cardano transaction types

Cardano's L1 transaction type (Chang 2 hardfork) served as an initial model for Midgard's L2 transaction type. In the following, field types prefixed by a question mark ? are set to appropriate "empty" defaults during deserialization if the serialized transaction omits them:[2] [3]

$$\mathsf{CardanoTx} := \left\{\begin{array}{ll} \mathsf{body} : & \mathsf{CardanoTxBody} \\ \mathsf{wits} : & \mathsf{CardanoTxWits} \\ \mathsf{is\_valid} : & \mathsf{Bool} \\ \mathsf{auxiliary\_data} : & ?\ \mathsf{TxMetadata} \end{array}\right\}$$

[2] In this section, we are mainly concerned with comparing Cardano and Midgard's deserialized data types. Serialization formats and conversions are addressed in Midgard's CDDL specifications.

[3] For simplicity of exposition, we omit the "era" type parameters, effectively coercing them all to the Conway era. We also simplify the script types.

$$
\text{CardanoTxBody} := \left\{ \begin{array}{ll}
\text{spend\_inputs} : & \text{Set}(\text{OutputRef}) \\
\text{collateral\_inputs} : & ?\ \text{Set}(\text{OutputRef}) \\
\text{reference\_inputs} : & ?\ \text{Set}(\text{OutputRef}) \\
\text{outputs} : & [\text{Output}] \\
\text{collateral\_return} : & ?\ \text{Output} \\
\text{total\_collateral} : & ?\ \text{Coin} \\
\text{certificates} : & ?\ [\text{Set}(\text{Certificate})] \\
\text{withdrawals} : & ?\ \text{Map}(\text{RewardAccount}, \text{Coin}) \\
\text{fee} : & \text{Coin} \\
\text{validity\_interval} : & \text{CardanoValidityInterval} \\
\text{required\_signer\_hashes} : & ?\ \text{Set}(\text{VKeyHash}) \\
\text{mint} : & ?\ \text{Value} \\
\text{script\_integrity\_hash} : & ?\ \text{Hash} \\
\text{auxiliary\_data\_hash} : & ?\ \text{Hash} \\
\text{network\_id} : & ?\ \text{Network} \\
\text{voting\_procedures} : & ?\ \text{VotingProcedures} \\
\text{proposal\_procedures} : & ?\ \text{Set}(\text{ProposalProcedure}) \\
\text{current\_treasury\_value} : & ?\ \text{Coin} \\
\text{treasury\_donation} : & ?\ \text{Coin}
\end{array} \right\}
$$

$$
\text{CardanoTxWits} := \left\{ \begin{array}{ll}
\text{addr\_tx\_wits} : & ?\ \text{Set}(\text{VKey}, \text{Signature}, \text{VKeyHash}) \\
\text{boot\_addr\_tx\_wits} : & ?\ \text{Set}(\text{BootstrapWitness}) \\
\text{script\_tx\_wits} : & ?\ \text{Map}(\text{ScriptHash}, \text{CardanoScript}) \\
\text{data\_tx\_wits} : & ?\ \text{TxDats} \\
\text{redeemer\_tx\_wits} : & ?\ \text{Redeemers}
\end{array} \right\}
$$

$$
\begin{aligned}
\text{CardanoValidityInterval} &:= (\text{Option}(\text{Slot}), \text{Option}(\text{Slot})) \\
\text{CardanoScript} &:= \text{TimelockScript}(\text{Timelock}) \\
&\mid \text{PlutusScript}(\text{CardanoPlutusVersion}, \text{PlutusBinary}) \\
\text{CardanoPlutusVersion} &:= \text{PlutusV1} \\
&\mid \text{PlutusV2} \\
&\mid \text{PlutusV3}
\end{aligned}
$$

1.6.2 Deviations from the Cardano transaction types

Midgard's transaction types deviate from Cardano's in the following ways:

No staking or governance actions. Midgard's consensus protocol is not based on Ouroboros proof-of-stake, and its governance protocol is not based on Cardano's hard-fork-combinator update mechanism. Furthermore, Midgard's L1 scripts cannot authorize arbitrary staking or governance actions on behalf of users. For this reason, staking and governance actions are prohibited in Midgard L2 transactions.

No pre-Conway features. Midgard does not need to maintain backwards compatibility with pre-Conway eras. Midgard deposits and transactions will be prohibited from using bootstrap addresses, and Shelley addresses with Plutus versions older than Plutus V3. Public key hash credentials, native scripts, and Plutus scripts at or above V3 are allowed.

No transaction metadata. Midgard prohibits using metadata in L2 transactions but allows users to set the `auxiliary_data_hash` to any arbitrary hash, which Midgard's ledger's rules

do not require to match the empty transaction metadata field. This preserves users' ability to pin metadata content via a hash in Midgard's ledger, but they are expected to store the actual metadata offchain.

No datum hashes in outputs. Midgard requires all utxo datums to be inline, which avoids the need to index the datum-hash to datum map from transactions' datum witnesses.

CIPs 112 and 128. Midgard has adopted CIP-112 and CIP 128 ahead of Cardano mainnet, which we expect to merge them in 2025. CIP-112 provides a new "Observe" script purpose, a more principled and streamlined alternative to the "Withdraw 0" trick widely used by most Cardano dApps. It also allows native scripts to conditionally forward their logic to an observer script. On the other hand, CIP-128 will significantly improve the execution efficiency achievable in Plutus contracts by preserving the order of inputs in a submitted transaction (instead of ordering them by `OutputRef`).

Different network ID. Midgard transactions and utxo addresses use a different network ID to distinguish them from their Cardano mainnet counterparts. Furthermore, we are considering a prefix to minting policy IDs to distinguish tokens deposited from L1 from tokens minted on L2.

Replace slots with POSIX timestamps. Midgard's consensus protocol does not use slots like Ouroboros. Furthermore, the L1 contracts that enforce Midgard's consensus protocol are evaluated in the Plutus script context, where slots are converted to Posix timestamps. For these reasons, Midgard uses POSIX timestamps instead of slots in L2 transactions.

IsValid tag is always True. Midgard's consensus protocol does not use the IsValid tag, so it is always set to `True`.

1.6.3 Midgard simplified transaction types

Midgard's actual transaction types are complicated by the need to optimize their traversal by Plutus scripts. As an intermediate step towards them, the following simplified types illustrate how the Cardano transaction types (Section 1.6.1) are modified by Midgard's deviations (Section 1.6.2). We use the empty set symbol ∅ to indicate fields required to be empty in Midgard transactions and use the star symbol ⋆ to indicate new or modified fields:

$$\text{MidgardSTx} := \left\{ \begin{array}{ll} \text{body} : & \text{MidgardSTxBody} \\ \text{wits} : & \text{MidgardSTxWits} \\ \text{is\_valid} : & \text{Bool} \\ \emptyset\ \text{auxiliary\_data} : & ?\ \text{TxMetadata} \end{array} \right\}$$

$$\text{MidgardSTxBody} := \left\{ \begin{array}{ll} \text{spend\_inputs} : & \text{Set}(\text{OutputRef}) \\ \varnothing\ \text{collateral\_inputs} : & ?\ \text{Set}(\text{OutputRef}) \\ \text{reference\_inputs} : & ?\ \text{Set}(\text{OutputRef}) \\ \text{outputs} : & [\text{Output}] \\ \varnothing\ \text{collateral\_return} : & ?\ \text{Output} \\ \varnothing\ \text{total\_collateral} : & ?\ \text{Coin} \\ \varnothing\ \text{certificates} : & ?\ [\text{Set}(\text{Certificate})] \\ \varnothing\ \text{withdrawals} : & ?\ \text{Map}(\text{RewardAccount}, \text{Coin}) \\ \text{fee} : & \text{Coin} \\ \star\ \text{validity\_interval} : & ?\ \text{MidgardSValidityInterval} \\ \star\ \text{required\_observers} : & ?\ [\text{ScriptCredential}] \\ \text{required\_signer\_hashes} : & ?\ [\text{VKeyCredential}] \\ \text{mint} : & ?\ \text{Value} \\ \text{script\_integrity\_hash} : & ?\ \text{Hash} \\ \text{auxiliary\_data\_hash} : & ?\ \text{Hash} \\ \text{network\_id} : & ?\ \text{Network} \\ \varnothing\ \text{voting\_procedures} : & ?\ \text{VotingProcedures} \\ \varnothing\ \text{proposal\_procedures} : & ?\ \text{Set}(\text{ProposalProcedure}) \\ \varnothing\ \text{current\_treasury\_value} : & ?\ \text{Coin} \\ \varnothing\ \text{treasury\_donation} : & ?\ \text{Coin} \end{array} \right\}$$

$$\text{MidgardSTxWits} := \left\{ \begin{array}{ll} \text{addr\_tx\_wits} : & ?\ \text{Set}(\text{VKey}, \text{Signature}, \text{VKeyHash}) \\ \varnothing\ \text{boot\_addr\_tx\_wits} : & ?\ \text{Set}(\text{BootstrapWitness}) \\ \text{script\_tx\_wits} : & ?\ \text{Map}(\text{ScriptHash}, \text{MidgardSScript}) \\ \varnothing\ \text{data\_tx\_wits} : & ?\ \text{TxDats} \\ \text{redeemer\_tx\_wits} : & ?\ \text{Redeemers} \end{array} \right\}$$

$$\text{MidgardSValidityInterval} := (\text{Option}(\text{PosixTime}), \text{Option}(\text{PosixTime}))$$

$$\text{MidgardSScript} := \text{TimelockScript}(\star\ \text{TimelockObserver}) \mid \text{PlutusScript}(\text{MidgardSPlutusVersion}, \text{PlutusBinary})$$

$$\text{MidgardSPlutusVersion} := \text{PlutusV3}$$

1.6.4 Midgard transaction types

Midgard's transaction types modify the above simplified types by replacing all variable-length fields with hashes (indicated by the letter $\mathcal{H}$ below). The data availability layer is responsible for confirming that the hashes correspond to their preimages, and DA fraud proofs can be submitted if this correspondence is violated.

$$\text{MidgardTx} := \left\{ \begin{array}{ll} \text{body} : & \mathcal{H}(\text{MidgardTxBody}) \\ \text{wits} : & \mathcal{H}(\text{MidgardTxWits}) \\ \text{validity} : & \text{MidgardTxValidity} \end{array} \right\}$$

$$\text{MidgardTxBody} := \left\{ \begin{array}{ll} \text{spend\_inputs} : & \mathcal{H}([\text{OutputRef}]) \\ \text{reference\_inputs} : & ?\ \mathcal{H}([\text{OutputRef}]) \\ \text{outputs} : & \mathcal{H}([\text{Output}]) \\ \text{fee} : & \text{Coin} \\ \text{validity\_interval} : & ?\ \text{MidgardSValidityInterval} \\ \text{required\_observers} : & ?\ \mathcal{H}([\text{ScriptCredential}]) \\ \text{required\_signer\_hashes} : & ?\ \mathcal{H}([\text{VKeyCredential}]) \\ \text{mint} : & ?\ \text{Value} \\ \text{script\_integrity\_hash} : & ?\ \text{Hash} \\ \text{auxiliary\_data\_hash} : & ?\ \text{Hash} \\ \text{network\_id} : & ?\ \text{Network} \end{array} \right\}$$

$$\text{MidgardTxWits} := \left\{ \begin{array}{ll} \text{addr\_tx\_wits} : & ?\ \mathcal{H}(\text{Set}(\text{VKey}, \text{Signature}, \text{VKeyHash})) \\ \text{script\_tx\_wits} : & ?\ \mathcal{H}(\text{Map}(\text{ScriptHash}, \text{MidgardSScript})) \\ \text{redeemer\_tx\_wits} : & ?\ \mathcal{H}(\text{Redeemers}) \end{array} \right\}$$

$$\begin{aligned} \text{MidgardTxValidity} := \ & \text{TxIsValid} \\ & \mid \text{NonExistentInputUtxo} \\ & \mid \text{InvalidSignature} \\ & \mid \text{FailedScript} \\ & \mid \text{FeeTooLow} \\ & \mid \text{UnbalancedTx} \\ & \mid \text{etc.}(\text{TODO}) \end{aligned}$$

1.7 Confirmed state

When a Midgard block becomes confirmed, a selection of its block header fields is split into two groups (discarding the rest). These groups are used to populate the fields of two record types in Midgard's confirmed state:

$$\text{ConfirmedState} := \left\{ \begin{array}{ll} \text{header\_hash} : & \text{HeaderHash} \\ \text{prev\_header\_hash} : & \text{HeaderHash} \\ \text{utxo\_root} : & \text{MPTR} \\ \text{start\_time} : & \text{PosixTime} \\ \text{end\_time} : & \text{PosixTime} \\ \text{protocol\_version} : & \text{Int} \end{array} \right\}$$

$$\text{Settlement} := \left\{ \begin{array}{ll} \text{deposit\_root} : & \text{MPTR} \\ \text{withdraw\_root} : & \text{MPTR} \\ \text{start\_time} : & \text{PosixTime} \\ \text{end\_time} : & \text{PosixTime} \\ \text{resolution\_claim} : & \text{Option}(\text{ResolutionClaim}) \end{array} \right\}$$

$$\text{ResolutionClaim} := \left\{ \begin{array}{ll} \text{time} : & \text{PosixTime} \\ \text{operator} : & \text{VerificationKey} \end{array} \right\}$$

At genesis, `ConfirmedState` is set as follows:

$$\text{genesisConfirmedState} := \left\{ \begin{array}{ll} \text{header\_hash} := & 0 \\ \text{prev\_header\_hash} := & 0 \\ \text{utxo\_root} := & \text{MPTR}_{\text{empty}} \\ \text{start\_time} := & \text{system\_start} \\ \text{end\_time} := & \text{system\_start} \\ \text{protocol\_version} := & 0 \end{array} \right\}$$

Midgard only stores the latest confirmed block's `ConfirmedState` on L1, always overwriting the previous one. By contrast, its `Settlement` never overwrites that of the previous block. Instead, the `Settlement` spins into a separate settlement node that users and operators can reference to process deposits and withdrawal requests.

The current operator can optimistically attach a resolution claim to any settlement node, indicating that all deposits and withdrawals in the node have been processed and that the node will be removed from the settlement queue at a given resolution time. The resolution time is set to the claim's attachment time, shifted forward by Midgard's `maturity_duration` protocol parameter, in order to provide an opportunity fraud proofs to be verified on L1 that disprove the operator's claim that the settlement node is resolved. The operator is slashed if their claim is disproved; otherwise, starting from the resolution time, the operator can remove the settlement node and recover its min-ADA and some fees for processing the deposits/withdrawals.

Chapter 2

User event protocol

The user event protocol controls how users interact with Midgard. It consists of three L1-based interactions and one L2-based interaction.

2.1 Deposit (L1)

A user deposits funds into Midgard by submitting an L1 transaction that performs the following:

1. Spend an input `l1_nonce`, which uniquely identifies this deposit transaction.
2. Register a staking script credential to witness the deposit. The script is parametrized by `l2_id` (which is Blake2b256 hash of `l1_nonce`), and the credential's purpose is to disprove the existence of the deposit whenever the credential is *not* registered.[1]
3. Total count of tokens in the deposit must not exceed `max_transfer_token_count`, a Midgard protocol pararmeter.
4. Mint a deposit auth token to verify the following datum:

$$\text{DepositDatum} := \left\{ \begin{array}{ll} \text{event} : & \text{DepositEvent}, \\ \text{inclusion\_time} : & \text{PosixTime}, \\ \text{witness} : & \text{ScriptHash}, \\ \text{refund\_address} : & \text{Address}, \\ \text{refund\_datum} : & \text{Option(Data)} \end{array} \right\}$$

5. Send the user's deposited funds to the Midgard deposit address, along with the deposit auth token and the above datum.

At the time of the L1 deposit transaction, the deposit's `inclusion_time` is set to the sum of the transaction's validity interval upper bound and the `event_wait_duration` Midgard protocol parameter. According to Midgard's ledger rules:

Deposit inclusion. A block header must include deposit events with inclusion times falling within the block header's event interval, and it must *not* include any other deposit events.

[1] Cardano's ledger lacks a more direct method to disprove the existence of a utxo to a Plutus script.

Furthermore, the state queue enforces that event intervals are adjacent and non-overlapping. Therefore, while operators continue committing valid block headers, every deposit is eventually included in the state queue.

On the other hand, suppose a fraud proof is verified on L1 to prove that a block header in the state queue has violated the deposit inclusion rule. When this block header and all its descendants are removed, the state queue enforces that the next committed block header's event interval must contain all of those removed block headers' event intervals. Therefore, Midgard's ledger rules require this new block header to include all deposit events that should have been included in the removed block headers.

The deposit's outcome is determined as follows:

- If the deposit event is included in a settlement queue node, then the deposit utxo can be absorbed into the Midgard reserve.
- If the deposit's inclusion time is within the confirmed header's event interval but not within the event interval of any settlement queue node, then the deposit utxo can be refunded to its user according to the `refund_address` and `refund_datum` fields.

The deposit's `witness` staking credential must be deregistered when the deposit utxo is spent.

2.1.1 Minting policy

The `deposit` minting policy is statically parametrized on the `hub_oracle` minting policy. It oversees correctness of datums, and registration/unregistration of events' corresponding witness staking scripts.

Authenticate Deposit. Properly record a new deposit event from L1 to L2. Conditions:

1. Let `l1_nonce` be the output reference of a utxo on L1 that is spent in the deposit transaction.
2. Let `l2_id` be the Blake2b256 hash of serialized `l1_nonce`.
3. An NFT with own policy ID and asset name of `l2_id` must be minted and included in the deposit utxo.
4. The witness staking script, instantiated with `l2_id` must be registered in the transaction.
5. The redeemer used for registering the witness script must be equal to the `deposit` policy ID.
6. Let `deposit_addr` be the address of the deposit contract from Midgard's hub oracle.
7. Deposit utxo must be produced at `deposit_addr`.
8. Total number of deposited tokens (including ADA) must not exceed `max_tokens_allowed_in_deposits`, a protocol parameter.
9. Recorded `l1_nonce` in the deposit event must be correct.
10. The deposit's `inclusion_time` must be equal to transaction's time-validity upper bound plus `event_wait_duration` Midgard protocol parameter.

11. The hash of the witness script must be correctly stored in the deposit datum.

Burn Deposit NFT. Oversee conclusion of a deposit event (i.e. either its transfer to reserve or its refund) by requiring its NFTs to be burnt. Conditions:

1. Let `l2_id` be the corresponding ID of the target deposit event, provided via the redeemer.
2. An NFT with own policy ID and asset name of `l2_id` must be burnt.
3. The witness staking script, instantiated with `l2_id` must be unregistered in the transaction.
4. The redeemer used for unregistering the witness script must be correct. Namely, it must be equal to the `deposit` policy ID.

2.1.2 Spending validator

The `deposit` spending validator is statically parametrized on the `hub_oracle` minting policy. It's responsible for concluding deposits, whether for transfering them to Midgard reserve, or refunding stranded deposits.

Transfer to Reserve. Transaction for moving funds of finalized deposit events into Midgard reserve. Conditions:

1. Let `settlement_node` be the referenced settlement node.
2. The deposit must be included in the deposit tree of `settlement_node`. This also implies the inclusion time of the deposit falls within the time interval of `settlement_node`.
3. Let `reserve_addr` be the address of the Midgard reserve retrieved from hub oracle.
4. The utxo produced at `reserve_addr` must contain all the tokens from the deposit utxo, except for the deposit NFT.
5. The `BurnEventNFT` endpoint of the deposit minting script must be invoked with the corresponding `l2_id` of the deposit utxo.
6. No datum and no reference script must be attached to the reserve utxo.

Refund. Conditions:

1. The `BurnEventNFT` endpoint of the deposit minting script must be invoked with the corresponding `l2_id` of the deposit utxo.
2. All the tokens included in the deposit utxo (except for the deposit NFT) must go to the `refund_address` specified in the deposit datum.
3. The datum attached to the produced utxo must also match the one stated in deposit datum.
4. - If inclusion time of the stranded deposit falls within the time range of an existing settlement node, refund can only be allowed if the settlement node's corresponding tree does not contain the deposit (proven by providing a non-membership proof).
   - Otherwise, if the inclusion time falls in one of the time gaps of settlement queue, refund request is considered valid by referencing the immediate settlement node and showing the inclusion time falls in the gap.

2.2 Transaction request (L2)

A user's primary method of transacting on the Midgard ledger is to submit an L2 transaction directly to the current operator via a web-based API endpoint. Operators are expected to serve such APIs in a publicly accessible and employ sufficient security techniques to mitigate denial-of-service.

Users' L2 transactions follow the data structure defined in Section 1.6.4. In particular, users can freely set transaction time-validity intervals according to their preferences. As long as there is a non-empty overlap between a transaction's time-validity interval and a block's event interval, the transaction can be included in the block. Since event intervals are adjacent to each other, a user's valid transaction request can only miss the ledger in two cases:

- The operator censors it (see mitigation in Section 2.3).
- The operator commits the last block overlapping with the transaction before receiving the transaction request.[2]

By contrast, L1 transactions on Cardano can fail in several other time-related ways. Ouroboros blocks only occupy every 20th one-second slot on average and have limited transaction capacity. This means that users often have to set longer transaction validity intervals than they want and wait for many block confirmations of their tx before relying on its outcomes.

However, while on Cardano L1 two transactions with non-overlapping validity intervals cannot be included in the same block, the analogous L2 transactions *can* be included in a Midgard block if its event interval overlaps each transaction. Thus, transaction validity intervals on Midgard define only the relation between transactions and blocks but do not necessarily imply a temporal precedence relation between transactions.

Which other information should we include here for L2 transaction requests?

TODO

[2] Note that this case does not preclude transactions with short time-validity intervals from succeeding. For example, a transaction post-dated a couple of minutes in the future can still be included in a block even if its validity interval's duration is one millisecond.

2.3 Transaction order (L1)

A user who wants to mitigate the risk of censorship by the current operator can submit an L2 transaction as an L1 transaction order. A transaction order is created by an L1 transaction that performs the following:

1. Spend an input `l1_nonce`, which uniquely identifies this transaction order.
2. Register a staking script credential to witness the transaction order. The staking script is parametrized by `l1_nonce`, and the credential's purpose is to disprove the existence of the transaction order whenever the credential is *not* registered.
3. Mint a transaction order token to verify the following datum:

$$\text{TxOrderDatum} := \left\{ \begin{array}{ll} \text{event} : & \text{TxOrderEvent}, \\ \text{inclusion\_time} : & \text{PosixTime}, \\ \text{witness} : & \text{ScriptHash}, \\ \text{refund\_address} : & \text{Address}, \\ \text{refund\_datum} : & \text{Option}(\text{Data}) \end{array} \right\}$$

4. Send min-ADA to the Midgard transaction order address, along with the transaction order token and the above datum.

At the time of the L1 transaction order, its `inclusion_time` is set to the sum of the L1 transaction's validity interval upper bound and the `event_wait_duration` Midgard protocol parameter. According to Midgard's ledger rules:

Transaction order inclusion. A block header must include transaction orders with inclusion times falling within the block header's event interval, and it must *not* include any other transaction orders.

The transaction order's `inclusion_time` must be within its L2 transaction's time-validity interval. This is guaranteed optimistically via the `validity` field of the underlying `MidgardTx`.

Analogously to deposits and withdrawals, transaction orders will eventually be included in the state queue, as long as operators continue committing valid block headers. Furthermore, if any blocks are removed from the state queue, any new committed block must include the transaction orders that should have been included in the removed blocks.

The transaction order fulfills its purpose when its inclusion time is within the confirmed header's event interval. Whether or not the outcome of the order's L2 transaction was merged into the confirmed state, nothing more can be achieved with the transaction order, and it can be refunded according to the `refund_address` and `refund_datum`.

The transaction order's `witness` staking credential must be deregistered when the order utxo is spent.

2.3.1 Minting policy

The `tx_order` minting policy is statically parametrized on the `hub_oracle` minting policy. It oversees correctness of datums, and registration/unregistration of events' corresponding witness staking scripts.

Authenticate Order. Properly record a new L2 transaction order order event on L1. Conditions:

1. Let `l1_nonce` be the output reference of a utxo on L1 that is spent in the order transaction.
2. Let `l1_id` be the Blake2b256 hash of serialized `l1_nonce`.
3. An NFT with own policy ID and asset name of `l1_id` must be minted and included in the transaction order utxo.
4. The witness staking script, instantiated with `l1_id` must be registered in the transaction.
5. The redeemer used for registering the witness script must be equal to the `tx_order` policy ID.
6. Let `tx_order_addr` be the address of the transaction order contract from Midgard's hub oracle.
7. Transaction order utxo must be produced at `tx_order_addr`.
8. The transaction order's `inclusion_time` must be equal to transaction's time-validity upper bound plus `event_wait_duration` Midgard protocol parameter.
9. The hash of the witness script must be correctly stored in the datum.

Burn Transaction Order NFT. Oversee the conclusion of transaction order by requiring its NFTs to be burnt. Conditions:

1. Let `l1_id` be the corresponding ID of the target transaction order, provided via the redeemer.
2. An NFT with own policy ID and asset name of `l1_id` must be burnt.
3. The witness staking script, instantiated with `l1_id` must be unregistered in the transaction.
4. The redeemer used for unregistrering the witness script must be correct. Namely, it must be equal to the `tx_order` policy ID.

2.3.2 Spending validator

The `tx_order` spending validator is statically parametrized on the `hub_oracle` minting policy. It's responsible for validating conclusion of transaction orders, or refunding stranded ones.

Conclude. Transaction for spending a confirmed transaction order. Conditions:

1. Let `settlement_node` be the referenced settlement node.
2. The transaction order must be included in the transactions tree of `settlement_node`. This also implies the inclusion time of the order falls within the time interval of `settlement_node`.
3. The `BurnEventNFT` endpoint of the transaction order minting script must be invoked with the corresponding `l1_id` of the order utxo.
4. Let `refund_address` and `refund_datum` be the refund information retrieved from the transaction order's datum.
5. The min-ADA of the transaction order utxo must go to `refund_address`.
6. The datum attached to this output utxo must be the same as `refund_datum`. **TODO**
7. No reference script must be attached to the output utxo.

Refund. Conditions:

1. The `BurnEventNFT` endpoint of the transaction order minting script must be invoked with the corresponding `l1_id` of the order utxo.
2. Let `refund_address` and `refund_datum` be the refund information retrieved from the transaction order's datum.
3. The min-ADA of the transaction order utxo must go to `refund_address`.
4. The datum attached to this output utxo must be the same as `refund_datum`.
5. - If inclusion time of the stranded transaction order falls within the time range of an existing settlement node, refund can only be allowed if the settlement node's corresponding tree does not contain the transaction order (proven by providing a non-membership proof).
   - Otherwise, if the inclusion time falls in one of the time gaps of settlement queue, refund request is considered valid by referencing the immediate settlement node and showing the inclusion time falls in the gap.

2.4 Withdrawal order (L1)

A user initiates a withdrawal from Midgard by first producing an authentic token at the *outbox* contract on L2. This ensures two things:

1. The utxo becomes unspendable on L2, until its corresponding withdrawal event removes it from ledger.
2. The number of tokens included for withdrawal does not exceed the `max_tokens_allowed` protocol parameter.

To actually bring the funds to L1, users must then submit an L1 transaction that performs the following:

1. Spend an input `l1_nonce`, which uniquely identifies this withdrawal order.
2. Register a staking script credential to witness the withdrawal order. The staking script is parametrized by `l1_id` (which is simply the hash of the serialized `l1_nonce`), and the credential's purpose is to disprove the existence of the withdrawal order whenever the credential is *not* registered.
3. Mint a withdrawal order token to verify the following datum:

$$\text{WithdrawalOrderDatum} := \left\{ \begin{array}{ll} \text{event} : & \text{WithdrawalEvent}, \\ \text{inclusion\_time} : & \text{PosixTime}, \\ \text{witness} : & \text{ScriptHash}, \\ \text{refund\_address} : & \text{Address}, \\ \text{refund\_datum} : & \text{Option}(\text{Data}) \end{array} \right\}$$

4. Send min-ADA to the Midgard withdrawal order address, along with the withdrawal order token and the above datum.

We could adapt Midgard's incentive structure to encourage operators to perform this second step on L1 on behalf of users, relieving users from this chore during normal protocol operation.

At the time of the L1 withdrawal order, its `inclusion_time` is set to the sum of the L1 transaction's validity interval upper bound and the `event_wait_duration` Midgard protocol parameter. According to Midgard's ledger rules:

Withdrawal order inclusion. A block header must include withdrawal orders with inclusion times falling within the block header's event interval, and it must *not* include any other withdrawal orders.

The withdrawal order's outcome is determined as follows:

- If the withdrawal event is included in a settlement queue node, its utxo can be transferred to the payout contract so that it can be funded from the Midgard reserve.
- If the withdrawal order's inclusion time is within the confirmed header's event interval but not within the event interval of any settlement queue node, then the withdrawal order utxo can be refunded to its user according to the `refund_address` and `refund_datum` fields.

The withdrawal order's `witness` staking credential must be deregistered when the withdrawal order utxo is spent.

2.4.1 Minting policy

The `withdrawal` minting policy is statically parametrized on the `hub_oracle` minting policy. It oversees correctness of datums, and registration/unregistration of events' corresponding witness staking scripts.

Authenticate Withdrawal. Properly record a new withdrawal order event from L2 to L1. Conditions:

1. Let `l1_nonce` be the output reference of a utxo on L1 that is spent in the order transaction.
2. Let `l1_id` be the Blake2b256 hash of serialized `l1_nonce`.
3. An NFT with own policy ID and asset name of `l1_id` must be minted and included in the withdrawal order utxo.
4. The witness staking script, instantiated with `l1_id` must be registered in the transaction.
5. The redeemer used for registering the witness script must be equal to the `withdrawal` policy ID.
6. Let `withdrawal_addr` be the address of the withdrawal contract from Midgard's hub oracle.
7. Withdrawal order utxo must be produced at `withdrawal_addr`.
8. The withdrawal order's `inclusion_time` must be equal to transaction's time-validity upper bound plus `event_wait_duration` Midgard protocol parameter.
9. The hash of the witness script must be correctly stored in the withdrawal datum.
10. Let `utxos_root` be the root of the latest ledger at the latest node of the state queue, and `l2_outref` be the output reference of the utxo subject to withdrawal.
11. Let `l2_output` be the utxo itself at `outbox` contract on L2, provided via the redeemer.
12. `utxos_root` must contain an item with key `l2_outref` and value `l2_output`.
13. Let `l2_value` be the recorded value on L2 stored in `WithdrawalInfo`.
14. `l2_value` and `value` from `l2_output` must be equal.
15. Address from `l2_output` must be the same as outbox contract's on L2.

Burn Withdrawal NFT. Oversee start of the funding for a withdrawal order by requiring its NFTs to be burnt, and having the utxo reproduced at the `payout` contract. Conditions:

1. Let `l1_id` be the corresponding ID of the target withdrawal order, provided via the redeemer.
2. An NFT with own policy ID and asset name of `l1_id` must be burnt.
3. The witness staking script, instantiated with `l1_id` must be unregistered in the transaction.
4. The redeemer used for unregistering the witness script must be correct. Namely, it must be equal to the `withdrawal` policy ID.

2.4.2 Spending validator

The `withdrawal` spending validator is statically parametrized on the `hub_oracle` minting policy. It's responsible for initializing the funding phase of a withdrawal order by reproducing its utxo at the `payout` contract, or refunding stranded withdrawal orders.

Initialize Payout. Transaction for reproducing the order at the payout contract to be filled up from the Midgard reserve. Conditions:

1. Let `settlement_node` be the referenced settlement node.
2. The withdrawal order must be included in the withdrawal tree of `settlement_node`. This also implies the inclusion time of the order falls within the time interval of `settlement_node`.
3. Let `payout_addr` be the address of the intermediary payout contract retrieved from hub oracle.
4. The utxo produced at `payout_addr` must have the same `value` as the withdrawal order utxo, without the withdrawal NFT, and with the `payout` NFT (same asset name) added.
5. The `BurnEventNFT` endpoint of the withdrawal minting script must be invoked with the corresponding `l1_id` of the withdrawal utxo.
6. The minting logic from `payout` must be invoked, minting an NFT with the same asset name as the withdrawal NFT being burnt in the transaction.
7. The datum attached to the payout accumulator utxo must hand over `l2_value`, `l1_address` and `l1_datum` unchanged.
8. No reference script must be attached to the payout accumulator utxo.

Refund. Conditions:

1. The `BurnEventNFT` endpoint of the withdrawal minting script must be invoked with the corresponding `l1_id` of the withdrawal order utxo.
2. The min-ADA included in the withdrawal order utxo must go to the `refund_address` specified in the withdrawal datum.
3. The datum attached to the produced utxo must also match the one stated in withdrawal datum.
4. - If inclusion time of the stranded withdrawal falls within the time range of an existing settlement node, refund can only be allowed if the settlement node's corresponding tree does not contain the withdrawal (proven by providing a non-membership proof).
   - Otherwise, if the inclusion time falls in one of the time gaps of settlement queue, refund request is considered valid by referencing the immediate settlement node and showing the inclusion time falls in the gap.

2.5 Witness Staking Script

A generic staking script on L1, parameterized by the unique identifier of a user event, whose registration state can be used to prove a falsely stated user event has not happened.

There are 3 logical endpoints this script needs to support:

1. Validating registration for an authentic user event.
2. Validating unregistration for the conclusion of a previously registered user event.
3. Allowing an isolated registration that is immediately followed by an unregistration in the same transaction.[3] Essentially proving the unregistered state of the staking script, without changing its state.

2.5.1 Staking script

The `witness` script is parameterized by the unique ID of a corresponding user event (deposit, withdrawal, or transaction order). Let `event_id` be this identifier, which is the Blake2b256 hash of an L1 nonce output-reference spent in the user event's first transaction.

Mint or Burn. Script invocation to be included in the first transaction of a user event. Conditions:

1. Let `target_policy` be the policy ID of the user event NFT, accessed via the redeemer.
2. Registration/unregistration of this witness script is respectively tied to the `target_policy`'s mint/burn logic.
   - Registration is allowed if a user event NFT with asset name of `event_id` is minted.
   - Unregistration is allowed if a user event NFT with asset name of `event_id` is burnt.

Register to Prove Not Registered. Redeemer for proving the unregistered state of the witness script instance. Conditions:

1. Let `redeemer_index` be the positional index of the current redeemer (`RegisterCredential` certificate) within the `redeemers` field of the transaction.
2. The redeemer pair that comes after `redeemer_index` must be of the same credential under an `UnregisterCredential` certificate.
3. Redeemer data of this following redeemer must be a `UnregisterToProveNotRegistered`, such that the index it carries is equal to `redeemer_index`.

Unregister to Prove Not Registered. Redeemer to follow `RegisterToProveNotRegistered`. Conditions:

1. Let `redeemer_index` be the positional index of the `RegisterToProveNotRegistered` redeemer (`RegisterCredential` certificate) within the `redeemers` field of the transaction.
2. Current certificate must be an `UnregisterCredential`, and its underlying credential must be equal to the one retrieved using `redeemer_index`.
3. Redeemer data at `redeemer_index` must be a `RegisterToProveNotRegistered`, such that the index it carries is equal to `redeemer_index`.

[3] This is possible because Cardano ledger allows custom ordering of publish redeemers (TODO?).

Chapter 3

Consensus protocol

This chapter describes Midgard's L1 contract-based consensus protocol, which establishes the canonical chain of valid blocks. It consists of the following components:

- The operator directory is an onchain data structure that tracks active, retired, and newly registered Midgard operators.
- The scheduler is an onchain mechanism that assigns evenly sized time windows to operators on a rotating schedule.
- The state queue is an onchain data structure that holds operators' committed block headers until they are merged into the confirmed state or disqualified by fraud proofs.
- When operators aren't committing blocks for a long time, the escape-hatch mechanism temporarily lowers the bond requirement for new operators.
- Computation threads facilitate the onchain validation of submitted fraud proofs, splitting it up into steps that fit within Cardano's limits on transaction size, computation, and memory.
- The fraud proof catalogue defines the universe of fraud proof verification procedures for which computation threads can be spawned to target a block header in the state queue.
- A fraud proof token indicates that a computation thread has successfully concluded to verify a fraud proof about a block header in the state queue.
- The Midgard hub oracle wires all the onchain components together by storing their minting policy IDs and spending validator addresses for easy reference.

3.1 Time model

Midgard partitions time in two different ways:

Operator shifts. Predefined, non-overlapping time intervals assigned to operators by the Midgard scheduler. Each operator has the exclusive privilege to commit blocks to the state queue and resolve nodes in the settlement queue during their assigned shifts. If an operator fails to commit blocks regularly in their shift, the next operator can take over.

Event intervals. Emergent, non-overlapping time intervals claimed by operators' committed blocks. Each operator block is expected to include all user events with inclusion times within its event interval and to exclude all other user events. For L1 user events (deposits, transaction orders, withdrawal orders), this is enforced by Midgard's ledger rules.

Operator shifts are evenly sized according to the `shift_duration` Midgard protocol parameter. The scheduler assigns operators to shifts by iterating over the list of active operators in key-descending order, allowing new operators into the list at the end of each cycle. The state queue enforces this schedule by requiring the time-validity upper bound of every block commitment transaction to fall within its operator's shift.

Each block's event interval is between the previous block's time-validity upper bound and its own time-validity upper bound. The block's time-validity lower bound is irrelevant to Midgard's consensus protocol because the causal relationship between blocks is determined by the forward links between state queue nodes, as well as the backwards links between block headers. An operator is free to submit blocks at a rapid cadence during their shift.[1]

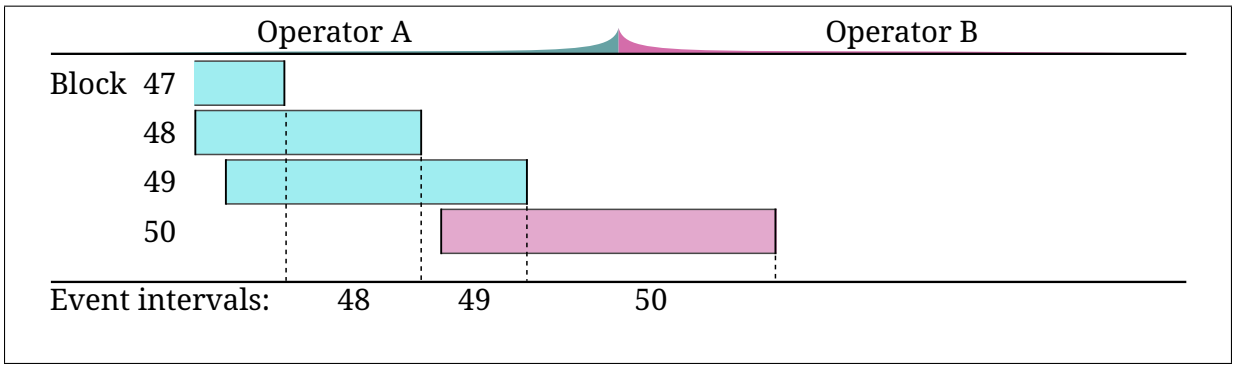

Figure 3.1.1: Midgard time model. Each rectangle shows a block's time-validity interval. The color indicates the operator who must have committed the block. The top axis shows the operator shifts. The bottom axis shows the event intervals.

[1] Indeed, an operator may commit blocks without waiting for L1 confirmation. As long as the blocks are valid, no one can interfere with the operator's chain of block commitment transactions. Moreover, if operators coordinate their activity offchain, this rapid cadence can be mostly maintained across shift boundaries.

3.2 Operator directory

Midgard operators receive and process L2 events from users, collate them into blocks, publish the full block contents on the data availability layer, and commit the block headers to Midgard's state queue on L1. This responsibility is shared among the operators on a rotating schedule, with each operator getting a turn to exclusively process L2 events and then commit their block of events at the end of their turn.

Anyone can register to become a Midgard operator if they post the required ADA bond. Registrants must wait a prescribed period of time before they activate as an operator. An active operator can retire to remove themself from the rotating schedule. Retired operators must wait until all their committed block headers mature before recovering their ADA bond.

An operator's ADA bond collateralizes their promise to faithfully process L2 events and commit valid blocks to L1. If an operator's block header is proven to be fraudulent, then the operator is disqualified and forfeits their bond. Similarly, an operator that submits duplicate registrations forfeits the bonds placed in the duplicates.

Every forfeited bond is split into a reward paid to the fraud prover and a slashing penalty paid (via transaction fees) to the Cardano treasury. Among the Midgard protocol parameters, the `fraud_prover_reward` and `slashing_penalty` parameters must sum up to the `required_bond` parameter.

3.2.1 Utxo representation

The Operator Directory keeps track of all Midgard operators and stores their bond deposits. It separates operators into three groups based on their status, with every group using operators' public key hashes (PKHs) as keys:

- `registered_operators` is a first-in-first-out (FIFO) queue that tracks operators who have posted their ADA bonds and are waiting to activate. It is implemented as a key-unordered linked list (see Section A.1.8), where each new registrant is prepended to the beginning of the list. This means that the earliest registrant to become eligible for activation is always at the last node of the list.
- `active_operators` is a set that tracks active operators participating in the rotating schedule, implemented as a key-ordered linked list (see Section A.1.9).
- `retired_operators` is a set that tracks retired operators waiting to recover their ADA bonds, implemented as a key-ordered linked list.

The Operator Directory keeps track of the activation time of every registered operator, which indicates when the operator will become eligible for activation. It also keeps track of any holds on the ADA bonds of active and retired operators, which prevent those operators from recovering their bonds until their latest committed blocks and latest settlement resolution claims mature.

$$\begin{aligned}
\mathsf{RegisteredOperator} &:= \left(\mathsf{activation_time} : \mathsf{PosixTime}\right) \\
\mathsf{ActiveOperator} &:= \left(\mathsf{bond_unlock_time} : \mathsf{Option}(\mathsf{PosixTime})\right) \\
\mathsf{RetiredOperator} &:= \left(\mathsf{bond_unlock_time} : \mathsf{Option}(\mathsf{PosixTime})\right)
\end{aligned}$$

The above are app-specific types for the `data` field of their respective lists' `NodeDatum`. For example, the `NodeDatum` of `registered_operators` is:

$$\text{RegisteredOperatorDatum} := \text{NodeDatum}(\text{RegisteredOperator})$$

3.2.2 Registered operators

The `registered_operators` queue keeps track of operators after registering and before activating or de-registering them.

Minting policy

The `registered_operators` minting policy implements state transitions for its key-unordered linked list. It is statically parametrized on the `hub_oracle` and `retired_operators` minting policies. Redeemers:

Init. Initialize the `registered_operators` queue via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle NFT.
2. The transaction must Init the `registered_operators` queue.

Register Operator. Prepend a node into the `registered_operators` queue with the operator's ADA bond at the operator's key, noting the registration time. Grouped conditions:

- Verify the registration and prepend the registrant to the registered operators:
  1. The transaction must Prepend a node into the `registered_operators` queue. Let that node be `registered_node`.
  2. The `key` field of `registered_node` must correspond to a public key hash that signed the transaction.
  3. The `activation_time` field of `registered_node` must equal the sum of the Midgard `registration_duration` protocol parameter and the upper bound of the transaction's validity interval.
  4. The Lovelace in the `registered_node` must equal the `required_bond` Midgard parameter.

Activate Operator. Transfer an operator node from the `registered_operators` queue to the `active_operators` set. Grouped conditions:

- Remove the registrant from the registered operators if it is eligible to activate:
  1. The transaction must Remove a node from the `registered_operators` queue. Let that node be `registered_node`.
  2. The lower bound of the transaction validity interval must meet or exceed the `activation_time`.
- Verify that the registrant is being added to active operators:
  3. The transaction must include the Midgard hub oracle NFT in a reference input.

4. Let `active_operators` be the policy ID in the corresponding field of the Midgard hub oracle.

5. The transaction must Insert a node into the `active_operators` set, as evidenced by minting a `active_operators` node NFT for the `registered_node` key.

- Verify that the registrant is not already retired:

6. The transaction must include a reference input of a `retired_operators` node that proves the `registered_node` key's non-membership in that list.

Deregister Operator. Remove a node from the `registered_operators` queue, with the operator's consent, and return the ADA bond to the operator. Conditions:

1. The transaction must Remove a node from the `registered_operators` queue. Let `registered_node` be that node.

2. The `key` field of `registered_node` must correspond to a public key hash that signed the transaction.

The operator consents to the transaction, so the ADA bond is assumed to be returned to the operator's control.

Remove Duplicate Slash Bond. Remove a node from the `registered_operators` queue if its key duplicates the key of any other node among the registered, active, or retired operators. Do *not* return the duplicate node's ADA bond to its operator. Conditions:

1. The transaction must Remove a node from the `registered_operators` queue. Let that node be `duplicate_node`.

2. The transaction fees must meet or exceed the `slashing_penalty` protocol parameter, denominated in Lovelaces.

3. Let `witness_status` be one of: `Registered`, `Active`, or `Retired`.

4. If `witness_status` is `Registered`:

   (a) The transaction must include a reference input of a `registered_operators` node that proves the `duplicate_node` key's membership in that list.

5. If `witness_status` is `Active`:

   (a) The transaction must include the Midgard hub oracle NFT in a reference input.

   (b) Let `active_operators` be the policy ID in the corresponding field of the Midgard hub oracle.

   (c) The transaction must include a reference input of a `active_operators` node that proves the `duplicate_node` key's membership in that list.

6. If `witness_status` is `Retired`:

   (a) The transaction must include a reference input of a `retired_operators` node that proves the `duplicate_node` key's membership in that list.

The submitter of the Remove Duplicate Slash Bond transaction is considered to be the fraud prover, so the conditions for that redeemer do not need to explicitly enforce that the `fraud_prover_reward` is paid out because the submitter consents to the transaction.

Spending validator

The spending validator of `registered_operators_addr` always forwards to its corresponding minting policy (statically parametrized) and requires the transaction to invoke it. It does not allow any in-place modifications to the `RegisteredOperator` value of the node `data` field. Conditions:

1. The transaction must mint or burn tokens of the `registered_operators` minting policy.

3.2.3 Active operators

The `active_operators` set keeps track of operators after activating and before slashing or retiring them.

Minting policy

The `active_operators` minting policy implements state transitions for its key-ordered linked list. It is statically parametrized on the `hub_oracle`, `registered_operators`, and `retired_operators` minting policies. Redeemers:

Init. Initialize the `active_operators` set via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle NFT.
2. The transaction must Init the `active_operators` set.

Activate Operator. Transfer an operator node from the `registered_operators` queue to the `active_operators` set. Conditions:

1. The transaction must Insert a node into the `active_operators` set. Let that node be `active_node`.
2. The `bond_unlock_time` field of `active_node` must be `None`.
3. The transaction must Remove a node from the `registered_operators` queue, as evidenced by burning a `registered_operators` node NFT corresponding to the `active_node` key.

Remove Operator Bad State. Remove an operator's node from the `active_operators` set without returning the operator's ADA bond to the operator, as a consequence of committing a fraudulent block to the state queue. Conditions:

1. Let `slashed_operator` be a redeemer argument indicating the operator being slashed.
2. The transaction must Remove a node from the `active_operators` set. Let that node be `removed_node`.
3. `slashed_operator` must match the key of `removed_node`.
4. The transaction fees must meet or exceed the `slashing_penalty` protocol parameter, denominated in Lovelaces.
5. The transaction must include the Midgard hub oracle NFT in a reference input.
6. Let `state_queue` be the policy ID in the corresponding field of the Midgard hub oracle.

7. The transaction must Remove a node from the `state_queue` via the Remove Fraudulent Block Header redeemer. The `fraudulent_operator` argument provided to that redeemer must match `slashed_operator`.

The state queue's onchain code is responsible for disposing of the operator's ADA bond.

Remove Operator Bad Settlement. Remove an operator's node from the `active_operators` set without returning the operator's ADA bond to the operator, as a consequence of attaching a fraudulent resolution claim to the settlement queue. Conditions:

1. Let `slashed_operator` be a redeemer argument indicating the operator being slashed.
2. The transaction must Remove a node from the `active_operators` set. Let that node be `removed_node`.
3. `slashed_operator` must match the key of `removed_node`.
4. The transaction fees must meet or exceed the `slashing_penalty` protocol parameter, denominated in Lovelaces.
5. The transaction must include the Midgard hub oracle NFT in a reference input.
6. Let `settlement_queue_addr` be the address in the corresponding field of the Midgard hub oracle.
7. The transaction must spend a node from the `settlement_queue_addr` via the Disprove Resolution Claim redeemer. The `fraudulent_operator` argument provided to that redeemer must match `slashed_operator`.

The settlement queue's onchain code is responsible for disposing of the operator's ADA bond.

Retire Operator. Transfer an operator node, unchanged, from the `active_operators` set to the `retired_operators` set. Conditions:

1. The transaction must Remove a node from the `active_operators` set. Let that node be `active_node`.
2. The transaction must Insert a node into the `retired_operators` set, as evidenced by minting a `retired_operators` node NFT corresponding to the `active_node` key.
3. Let `retired_node` be the node inserted into the `retired_operators` set.
4. The `bond_unlock_time` must match between `active_node` and `retired_node`.

Spending validator

The spending validator of `active_operators_addr` forwards to its corresponding minting policy (statically parametrized) when the transaction invokes it. When the minting policy isn't invoked, the spending validator updates the bond unlock time of an operator that commits a new block to the state queue or attaches a resolution claim to a settlement queue node. Redeemers:

List State Transition. Forward to minting policy. Conditions:

1. The transaction must mint or burn tokens of the `active_operators` minting policy.

Update Bond Hold New State. Update an operator's bond unlock time when they commit a block to the state queue. Grouped conditions:

- Update the bond unlock time of a operator:

  1. The transaction must *not* mint or burn tokens of the `active_operators` minting policy.
  2. Let `active_node` be an output of the transaction indicated by a redeemer argument.
  3. `active_node` must be an `active_operators` node that matches the datum argument of the spending validator on the `key` and `link` fields.
  4. The `bond_unlock_time` field of `active_node` must match the sum of the Midgard `maturity_duration` parameter and the upper bound of the transaction validity interval.

- Verify that the operator is currently committing a block header to the state queue:

  5. The transaction must include the Midgard hub oracle NFT in a reference input.
  6. Let `state_queue` be the policy ID in the corresponding field of the Midgard hub oracle.
  7. The transaction must Append a node into the `state_queue` via the Commit Block Header redeemer. The redeemer's `operator` field must match the `key` field of the `active_node`.

Update Bond Hold New Settlement. Update an operator's bond unlock time when they attach a resolution claim to a settlement node. Grouped conditions:

- Update the bond unlock time of a operator:

  1. The transaction must *not* mint or burn tokens of the `active_operators` minting policy.
  2. Let `active_node` be an output of the transaction indicated by a redeemer argument.
  3. `active_node` must be an `active_operators` node that matches the datum argument of the spending validator on the `key` and `link` fields.
  4. The `bond_unlock_time` field of `active_node` must match the sum of the Midgard `maturity_duration` parameter and the upper bound of the transaction validity interval.

- Verify that the operator is currently committing a block header to the state queue:

    5. The transaction must include the Midgard hub oracle NFT in a reference input.

    6. Let `settlement_queue_addr` be the policy ID in the corresponding field of the Midgard hub oracle.

    7. The transaction must spend a settlement queue node with the Attach Resolution Claim redeemer. The redeemer's `operator` field must match the `key` field of the `active_node`.

3.2.4 Retired operators

The `retired_operators` set keeps track of operators after retiring and before slashing or returning their ADA bonds.

Minting policy

The `retired_operators` minting policy implements structural operations for its key-ordered linked list. It is statically parametrized on the `hub_oracle` minting policy. Redeemers:

Init. Initialize the `retired_operators` set via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle NFT.

2. The transaction must Init the `retired_operators` set.

Retire Operator. Transfer an operator node, unchanged, from the `active_operators` set to the `retired_operators` set. Conditions:

1. The transaction must Insert a node into the `retired_operators` set. Let that node be `retired_node`.

2. The transaction must include the Midgard hub oracle NFT in a reference input.

3. Let `active_operators` be the policy ID in the corresponding field of the Midgard hub oracle.

4. The transaction must Remove a node from the `active_operators` set, as evidence by the burning of a `active_operators` node NFT corresponding to the `retired_node` key.

The active operators' minting policy ensures that the operator node's contents remain unchanged during the transfer.

Recover Operator Bond. Remove an operator's node from the `retired_operators` set, with the operator's consent, and return the ADA bond to the operator. Grouped conditions:

1. The transaction must Remove a node from the `retired_operators` set. Let that node be `retired_node`.

2. If the `bond_unlock_time` field of `retired_node` is *not* `None`, then the lower bound of the transaction validity interval must meet or exceed the `bond_unlock_time`.

The operator consents to the transaction, so the ADA bond is assumed to be returned to the operator's control.

Remove Operator Bad State. Remove an operator's node from the `retired_operators` set without returning the operator's ADA bond to the operator. Conditions:

1. Let `slashed_operator` be a redeemer argument indicating the operator being slashed.
2. The transaction must Remove a node from the `retired_operators` set. Let that node be `removed_node`.
3. `slashed_operator` must match the key of `removed_node`.
4. The transaction fees must meet or exceed the `slashing_penalty` protocol parameter, denominated in Lovelaces.
5. The transaction must include the Midgard hub oracle NFT in a reference input.
6. Let `state_queue` be the policy ID in the corresponding field of the Midgard hub oracle.
7. The transaction must Remove a node from the `state_queue` via the Remove Fraudulent Block Header redeemer. The `fraudulent_operator` argument provided to that redeemer must match `slashed_operator`.

The state queue's onchain code is responsible for paying out the fraud prover's reward from the operator's forfeited ADA bond.

Remove Operator Bad Settlement. Remove an operator's node from the `retired_operators` set without returning the operator's ADA bond to the operator, as a consequence of attaching a fraudulent resolution claim to the settlement queue. Conditions:

1. Let `slashed_operator` be a redeemer argument indicating the operator being slashed.
2. The transaction must Remove a node from the `retired_operators` set. Let that node be `removed_node`.
3. `slashed_operator` must match the key of `removed_node`.
4. The transaction fees must meet or exceed the `slashing_penalty` protocol parameter, denominated in Lovelaces.
5. The transaction must include the Midgard hub oracle NFT in a reference input.
6. Let `settlement_queue_addr` be the address in the corresponding field of the Midgard hub oracle.
7. The transaction must spend a node from the `settlement_queue_addr` via the Disprove Resolution Claim redeemer. The `fraudulent_operator` argument provided to that redeemer must match `slashed_operator`.

The settlement queue's onchain code is responsible for disposing of the operator's ADA bond.

Spending validator

The spending validator of `retired_operators_addr` always forwards to its corresponding minting policy (statically parametrized) and requires the transaction to invoke it. It does not allow any in-place modifications to the `RegisteredOperator` value of the node `data` field. Conditions:

1. The transaction must mint or burn tokens of the `retired_operators` minting policy.

3.3 Scheduler

The Midgard scheduler is an L1 mechanism that indicates which operator is assigned to the current shift and controls the transitions to the next shift and next operator. Whenever a shift ends, the next operator in key-descending order from the `active_operators` list has the exclusive privilege to advance the scheduler's state to the next shift, assigning it to themself.

When the root node of `active_operators` is reached, the highest-key active operator rewinds the scheduler to start a new cycle. However, the scheduler requires the `registered_operators` queue to be checked to see if any registered operators are eligible to activate. If so, all eligible registered operators must activate before the new cycle can begin.

Active operators can retire anytime without waiting for the end of the scheduler cycle. The scheduler automatically adjusts to retirement because the next operator is always selected among active operators. However, to minimize disruption, an operator should complete their shift before retiring.

We should allow operators to take over for negligent operators.
If a shift's assigned operator neglects their duty to commit blocks regularly to the state queue, the next active operator can take over the shift (without starting the next shift). Similarly, the next operator can advance the scheduler and take over if their predecessor neglects to do so. Midgard's `operator_inactivity_timeout` protocol parameter controls when this unscheduled takeover can happen.

TODO

3.3.1 Utxo representation

The scheduler state consists of a single utxo that holds the scheduler NFT, minted when Midgard is initialized via the hub oracle. That utxo's datum type is as follows:

$$\text{SchedulerDatum} := \left\{ \begin{array}{ll} \text{operator} : & \text{PubKeyHash} \\ \text{shift\_start} : & \text{PosixTime} \end{array} \right\}$$

The shift's inclusive lower bound is `shift_start`, and its exclusive upper bound is the sum of `shift_start` and the `shift_duration` Midgard protocol parameter.

3.3.2 Minting policy

The `scheduler` minting policy initializes the scheduler state. It is statically parametrized on the `hub_oracle` minting policy. Redeemers:

Init. Initialize the `scheduler` via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle token.
2. The transaction must mint the `scheduler` NFT.

3.3.3 Spending validator

The spending validator of `scheduler_addr` controls the evolution of the scheduler state. It is statically parametrized on the `active_operators` and `registered_operators` minting policies. Redeemers:

Advance. The next operator in key-descending order advances the scheduler to the next shift and assigns it to themself. Grouped conditions:

- Parse scheduler input and output:
  1. Let `scheduler_datum` be the datum argument for the spending validator being evaluated. It is assumed to belong to the transaction input that holds the scheduler NFT.
  2. Let `scheduler_output` be the transaction output that holds the scheduler NFT.
  3. Let `operator` be the `operator` field value of `scheduler_output`.
  4. Let `previous_operator` be the `operator` field value of `scheduler_datum`.
- Verify the shift transition:
  5. The shift interval of `scheduler_output` must equal that of `scheduler_datum`, moved forward by `shift_duration`.
- Verify operator consent:
  6. Either of the following must hold:
     (a) The transaction is signed by `operator`, and the `scheduler_output` shift interval contains the transaction validity interval.
     (b) The `scheduler_output` shift interval occurs entirely before the transaction validity interval.
- Verify the operator transition:
  7. The transaction must include a reference input of an `active_operators` node. Let `operator_node` be that node.
  8. The `key` of `operator_node` must match `operator`.
  9. The `link` of `operator_node` must match or exceed `previous_operator`.[2]

Rewind. The highest-key operator advances the scheduler to the next shift and assigns it to themself, provided that no registered operators are eligible to activate. Grouped conditions:

- Parse scheduler input and output:
  1. Let `scheduler_datum` be the datum argument for the spending validator being evaluated. It is assumed to belong to the transaction input that holds the scheduler NFT.
  2. Let `scheduler_output` be the transaction output that holds the scheduler NFT.

[2] It will exceed the previous operator if they have retired.

3. Let `operator` be the `operator` field value of `scheduler_output`.

4. Let `previous_operator` be the `operator` field value of `scheduler_datum`.

- Verify the shift transition:

5. The shift interval of `scheduler_output` must equal that of `scheduler_datum`, moved forward by `shift_duration`.

- Verify operator consent:

6. Either of the following must hold:

   (a) The transaction is signed by `operator`, and the `scheduler_output` shift interval contains the transaction validity interval.

   (b) The `scheduler_output` shift interval occurs entirely before the transaction validity interval.

- Rewind to a new operator cycle:

7. The transaction must include a reference input of the last `active_operators` node. Let `operator_node` be that node.

8. The transaction must include a reference input of the root `active_operators` node. Let `root_node` be that node.

9. The `key` of `operator_node` must match `operator`.

10. The `link` of `root_node` must match or exceed `previous_operator`.[3]

- Verify that no registered operator is eligible to activate:

11. The transaction must include a reference input of a `registered_operators` node. Let `registered_node` be that node.

12. `registered_node` must be the last node of the `registered_operators` queue. This means it corresponds to the *earliest* registrant that hasn't yet activated because new registrants are prepended to the beginning of that queue.

13. `registered_node` is *not yet* eligible for activation — the upper bound of the transaction validity interval is smaller than the `activation_time` of the `registered_node`.

[3] In other words, this means that there is no `active_operators` node with a smaller key than `previous_operator`. If there were such a node, then we would advance the scheduler to that node instead of rewinding.

3.4 State queue

The state queue is an L1 data structure that stores Midgard operators' committed block headers until they are confirmed. Active operators from the operator directory (see Section 3.2) take turns committing block headers to the state queue according to the rotating schedule enforced by the scheduler (see Section 3.3).

3.4.1 Utxo representation

The `state_queue` is implemented as a key-unordered linked list of block headers (see Section 1.1). Each `state_queue` node's key is its block header's hash.

$$\mathsf{StateQueueDatum} := \mathsf{NodeDatum}(\mathsf{Header})$$

$$\mathsf{valid\_key}(\mathsf{scd} : \mathsf{StateQueueDatum}) := \Big(\mathsf{scd.key} \equiv \mathsf{Some}(\mathsf{hash}(\mathsf{scd.data}))\Big)$$

Committing a block header to the `state_queue` means appending a node containing the block header to the end of the queue. After staying there for the `maturity_duration` (a protocol parameter), it is merged to the confirmed state (held at the `state_queue` root node) in the first-in-first-out (FIFO) order.

3.4.2 Minting policy

The `state_queue` minting policy controls the structural changes to the state queue. It is statically parametrized on the `hub_oracle`, `active_operators`, `retired_operators`, `scheduler`, and `fraud_proof` minting policies. Redeemers:

Init. Initialize the `state_queue` via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle token.
2. The transaction must Init the `state_queue`.

Commit Block Header. An operator commits a block header to the state queue if it is the operator's turn according to the rotating schedule. Grouped conditions:

- Commit the block header to the state queue, with the operator's consent:
  1. Let `operator` be a redeemer argument indicating the key of the operator committing the block header.
  2. The transaction must be signed by `operator`.
  3. The transaction must Append a node to the `state_queue`. Among the transaction outputs, let that node be `header_node` and its predecessor node be `previous_header_node`.
  4. `operator` must be the operator of `header_node`.
  5. The `key` field of `header_node` must be the hash of its `data` field.
- Verify that it is the operator's turn to commit according to the scheduler:

6. The transaction must include a reference input with the scheduler NFT. Let that input be the `scheduler_state`.

7. The `operator` field of `scheduler_state` must match `operator`.

- Verify block timestamps:

8. The `start_time` of `header_node` must be equal to the `end_time` of the `previous_header_node`.

9. The `end_time` of `header_node` must match the transaction's time-validity upper bound.

10. The `end_time` of `header_node` must be within the shift interval of `scheduler_state`.

- Update the operator's timestamp in the active operators set:

11. The transaction must include an input, spent via the Update Bond Hold New State redeemer, of an `active_operators` node with a key matching the `operator`.

Merge To Confirmed State. If a block header is mature, merge it to the confirmed state of the `state_queue`. Conditions:

1. The transaction must Remove a node from the `state_queue`. Let `header_node` be the removed node, `confirmed_state_node_input` be its predecessor node before removal, and `confirmed_state_node_output` be the remaining node after removal.

2. `confirmed_state_node_input` and `confirmed_state_node_output` must both be root nodes of `state_queue`.

3. `header_node` must be mature — the lower bound of the transaction validity interval meets or exceeds the sum of the `end_time` field of `header_node` and the Midgard `maturity_duration` protocol parameter.

4. `confirmed_state_node_output` must match:

   (a) `confirmed_state_node_input` on `start_time`.

   (b) `header_node` on `prev_header_hash`, `utxo_root`, `end_time`, and `protocol_version`.

5. The `header_hash` of `confirmed_state_node_output` must match the `header_node` key.

6. If either the `deposits_root` or `withdrawals_root` of `header_node` is *not* the MPT root hash of the empty set, a `settlement_queue` node must be appended via the New Settlement redeemer. The redeemer must mention `header_node` by input index.

Remove Fraudulent Block Header. Remove a fraudulent block header from the state queue and slash its operator's ADA bond. Grouped conditions:

- Remove the fraudulent block header:

1. Let `fraudulent_operator` be a redeemer argument indicating the operator who committed the fraudulent block header.

2. The transaction must Remove a node from the `state_queue`. Let `removed_node` be the removed node and `predecessor_node` be its predecessor node before removal.

3. `fraudulent_operator` must match the key of `removed_node`.

- Slash the fraudulent operator:

4. Let `operator_status` be a redeemer argument indicating whether the fraudulent operator is active or retired.

5. If `operator_status` is active:

   (a) The transaction must Remove a node from the `active_operators` set via the Remove Operator Bad State redeemer. The `slashed_operator` argument provided to that redeemer must match `fraudulent_operator`.

6. Otherwise:

   (a) The transaction must Remove a node from the `retired_operators` set via the Remove Operator Bad State redeemer. The `slashed_operator` argument provided to that redeemer must match `fraudulent_operator`.

- Verify that fraud has been proved for the removed node or its predecessor:

7. The transaction must include a reference input holding a `fraud_proof` token.

8. Let `fraud_proof_block_hash` be the last 28 bytes of the `fraud_proof` token.

9. One of the following must be true:

   (a) `fraud_proof_block_hash` matches the `predecessor_node` key. This means that a child of the fraudulent node is being removed.

   (b) `fraud_proof_block_hash` matches the `removed_node` key, and the last node of `state_queue` is `removed_node`. This means that the fraudulent node is being removed and has no more children.

3.4.3 Spending validator

The spending validator of `state_queue_addr` always forwards to its corresponding minting policy (statically parametrized) and requires the transaction to invoke it. It does not allow any in-place modifications to the `data` field of nodes in `state_queue`. Conditions:

1. The transaction must mint or burn tokens of the `state_queue` minting policy.

3.5 Escape hatch

The Midgard escape hatch is an L1 mechanism that can be triggered if the last block in the state queue is too old relative to the current time, as controlled by the Midgard `escape_hatch_block_age` parameter. In other words, it is triggered in the extreme case when the Midgard active operators have completely stopped committing blocks to the state queue.

When the escape hatch is triggered, Midgard enters an emergency period with a duration controlled by the Midgard `emergency_duration` parameter. During this emergency period, the Operator Directory allows new operators to register with a reduced bond requirement. As a result, honest actors can more easily become operators to restart Midgard block production.

When the emergency period is over, the reduced-bond operators have a grace period to voluntarily either retire or increase their bonds to remain as regular operators. After the grace period, anyone can retire any remaining operator that has not increased their bond, applying a minor penalty to the bond and claiming it as a reward. See Section 3.2.

3.5.1 Utxo representation

The Midgard escape hatch is represented onchain by a single utxo holding an NFT with this datum:

$$\text{EscapeHatchDatum} := \{\ \text{emergency\_end\_time} :\ \ \text{PosixTime}\ \}$$

This datum is set to zero when Midgard is initialized. When the escape hatch is triggered, it is set to the time when the emergency period will end.

3.5.2 Minting policy

The `escape_hatch` minting policy initializes the escape hatch state. It is statically parametrized on the `hub_oracle` minting policy. Redeemers:

Init. The transaction must mint the Midgard hub oracle token.

3.5.3 Spending validator

The spending validator of `escape_hatch_addr` controls the escape hatch trigger. Redeemers:

Trigger Escape Hatch. Conditions:

1. Let `last_header_node` be the last node in the `state_queue`.
2. The transaction's time-validity lower bound must exceed `last_header_node.end_time` by `escape_hatch_block_age`.
3. The transaction's time-validity interval duration must not exceed `escape_hatch_trigger_duration`.
4. Let `escape_hatch_output` be the transaction output that holds the `escape_hatch` token.
5. The `emergency_end_time` of `escape_hatch_output` must match the transaction's time-validity upper bound, moved forward by `emergency_duration`.

3.6 Settlement queue

The settlement queue tracks confirmed deposits, withdrawal orders, and transaction orders until they are spent. Each settlement node in the queue is created whenever a block merges into the confirmed state and includes any deposits, withdrawals, or transaction orders. The settlement node contains the block's deposits root hash, withdrawals root hash, transactions root hash, and event interval. The transactions root hash stored in the settlement node is the Merkle root of all transactions in the block, including both transaction requests (from operators' mempools) and transaction orders (from L1 events). This allows operators to resolve transaction order event UTxOs after merging their associated blocks into confirmed state. Users must reference a settlement node to spend its confirmed deposits, withdrawal orders, or transaction orders.

The current operator can optimistically resolve any settlement node, claiming that its confirmed deposits, withdrawal orders, and transaction orders have all been spent. To do so, the operator attaches a resolution claim to the settlement queue for the `maturity_duration` (a protocol parameter). During this maturity period, if the operator's resolution claim is proven fraudulent, the prover can slash the operator's bond and remove the resolution claim from the settlement node. After the maturity period, the settlement node can be removed from the queue if the resolution claim is intact.

> Perhaps resolving a settlement node should also release the user fees collected from the node's deposits, withdrawals, and transaction orders to the operator. These fees would incentivize the operator to resolve settlement nodes promptly.

The timestamp in the operator's node in the Operator Directory should be updated whenever an operator optimistically resolves a settlement node, similar to how it is updated when committing a block to the state queue. Updating this timestamp ensures that the operator cannot recover their bond until all their resolution claims have matured in the settlement queue.

3.6.1 Linked list representation

The settlement queue's nodes are arranged as an L1 linked list in the same chronological order as their corresponding confirmed blocks. This chronological order makes it possible to prove that an unspent L1 event is *stranded*—it has not been included in any existing confirmed block but cannot be included in any new block.

For example, suppose an operator block fraudulently excludes an L1 deposit, but no fraud prover removes the block before its confirmation. In that case, the deposit cannot be included in any subsequent block or spent into the reserves. Without the refund mechanism based on proof of stranding, the deposit would stay stranded forever at Midgard's deposit address.

An L1 event is stranded (and can therefore be refunded) if it satisfies all of the following conditions:

1. The event's inclusion time is earlier than the confirmed state's end time.
2. Any of the following conditions hold:
   (a) The event's inclusion time is *not* within the event interval of any settlement node.
   (b) The event's inclusion time is within the event interval of a settlement node. However, the event is *not* in the corresponding deposit or withdrawal tree in the settlement node.

In this way, while the Midgard protocol prevents deposits from ever being stranded in the first place (as long as fraud proofs are promptly submitted), the refund mechanism ensures that they can still be retrieved if they occur.

3.6.2 Minting policy

The `settlement_queue` minting policy is statically parametrized on the `hub_oracle` minting policy. It's responsible for appends to the `settlement_queue` list's end, and also removal of nodes anywhere in the list.

Init. Initialize the `settlement_queue` via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle NFT.
2. The transaction must Init the `settlement_queue` list.

Append Settlement Node. Append a settlement node to store a merged block's deposits, withdrawals and transaction orders. Conditions:

1. The transaction must include the Midgard hub oracle NFT in a reference input.
2. Let `state_queue` be the policy ID in the corresponding field of the Midgard hub oracle.
3. The transaction must Remove a `state_queue` node via the Merge To Confirmed State redeemer. Let `merged_block` be the block being merged to the confirmed state, and let `header_hash` be its header-hash key.
4. The transaction must Append a node to the settlement queue with a key matching `header_hash`. Let `new_settlement_node` be the node being appended.
5. `merged_block` and `new_settlement_node` must match on all of these fields:
   - `deposits_root`
   - `withdrawals_root`
   - `transactions_root`
   - `start_time`
   - `end_time`
6. The `resolution_claim` of `new_settlement_node` must be empty.

Resolve Settlement Node. Remove a settlement node if its resolution claim has matured. Conditions:

1. The transaction must Remove a node from the `settlement_queue`. Let `removed_settlement_node` be this node.
2. The `resolution_claim` of `removed_settlement_node` must not be empty.
3. The transaction must be signed by the `operator` of the `resolution_claim`.
4. The transaction's time-validity lower bound must match or exceed the `resolution_time` of the `resolution_claim`.

3.6.3 Spending validator

The spending validator of `settlement_queue` is statically parametrized on the `settlement_queue` and `hub_oracle` minting policy. Conditions:

List State Transition. Forward to minting policy. Conditions:

1. The transaction must mint or burn tokens of the `settlement_queue` minting policy.

Attach Resolution Claim. The current operator attaches a resolution claim to a settlement node. Conditions:

1. The spent input must be a settlement node without a resolution claim.
2. The spent input must be reproduced as a settlement node with a resolution claim.
3. The transaction must be signed by the resolution claim's operator.
4. The transaction must include the Midgard hub oracle NFT in a reference input.
5. Let `active_operators` and `scheduler` be the corresponding policy IDs in the Midgard hub oracle.
6. The transaction must include an input (`operator_node`), spent via the Update Bond Hold New Settlement redeemer, of an `active_operators` node with a key matching the resolution claim's operator.
7. The `bond_unlock_time` of the `operator_node` must match the `resolution_time` of the resolution claim.
8. The transaction must include the `scheduler` utxo as a reference input, indicating that the current operator matches the resolution claim's operator.

Disprove Resolution Claim. Disprove and detach a settlement node's resolution claim, slashing the claimant operator. Conditions:

1. The spent input must be a settlement node with a resolution claim. Let `operator` be the operator of that claim.
2. The spent input must be reproduced as a settlement node without a resolution claim.
3. The transaction must include the Midgard hub oracle NFT in a reference input.
4. Let `active_operators`, `deposit`, `withdrawal`, and `tx_order` be the corresponding policy IDs in the Midgard hub oracle.
5. The transaction must include either a deposit, a withdrawal, or an transaction order as a reference input. Let that reference input be `unprocessed_event`.
6. A valid membership proof must be provided, proving that `unprocessed_event` is a member of the corresponding tree in the settlement node.
7. The transaction's time-validity upper bound must be earlier than the resolution claim's `resolution_time`.
8. Let `operator_status` be a redeemer argument indicating whether `operator` is active or retired.

9. If `operator_status` is active:

    (a) The transaction must Remove a node from the `active_operators` set via the Remove Operator Bad Settlement redeemer. The `slashed_operator` argument provided to that redeemer must match `operator`.

10. Otherwise:

    (a) The transaction must Remove a node from the `retired_operators` set via the Remove Operator Bad Settlement redeemer. The `slashed_operator` argument provided to that redeemer must match `operator`.

3.7 Reserve and payout

Midgard's reserve safeguards funds absorbed from confirmed deposits until Midgard's payout accumulator utxos collect them. Midgard's payout accumulators collect funds from Midgard's reserve in order to pay out confirmed withdrawal orders in full.

When a deposit is absorbed into Midgard's reserve, its funds (excluding the deposit ID token) are sent to a single utxo at the reserve address without a datum, without a reference script attached, and without being merged with the funds from any other deposits (see Section 2.1.2). In other words, the deposit utxo is reproduced one-to-one at the reserve address after removing the datum and ID token that identified the deposit.

Each payout accumulator utxo is initialized by proving that its corresponding withdrawal order is confirmed (via a settlement node), burning the withdrawal order's ID token, and minting the corresponding payout ID token (see Section 2.4.2). The payout accumulator's datum is:

$$\text{PayoutDatum} := \left\{ \begin{array}{ll} \text{l1\_address} : & \text{Address} \\ \text{l1\_datum} : & \text{Option(Data)} \\ \text{l2\_value} : & \text{Value} \end{array} \right\}$$

The payout accumulator collects funds from the reserve until it contains sufficient funds for the withdrawal's payout, sending any excess funds back to the reserve. When the payout accumulator contains sufficient funds, it sends the funds to the withdrawal's destination address and datum.

3.7.1 Reserve spending validator

The spending validator of `reserve_addr` implements the logic of Midgard's reserve. Redeemers:

Spend. Spend a reserve utxo to transfer some of its funds into a payout accumulator.

1. The transaction must spend an input from the `payout_addr` via the `Collect_Reserve_Funds` redeemer.

3.7.2 Payout minting policy

The `payout` minting policy ensures that only a single payout accumulator can be initialized or payout can be completed per transaction. Redeemers:

Mint. Mint a single payout token. Conditions:

1. The transaction must mint exactly one token of `payout`.
2. The transaction must burn exactly one token of `withdrawal`.
3. The minted and burned tokens must match on token name.
4. The transaction must not mint or burn any other tokens.

Burn. Burn a single payout token. Conditions:

1. The transaction must burn exactly one token of `payout`.
2. The transaction must not mint or burn any other tokens.

3.7.3 Payout spending validator

The spending validator of `payout_addr` is responsible for collecting funds from the Midgard reserve until a complete payout can be sent to its destination.

Collect Reserve Funds. Collect funds from the Midgard reserve. Conditions:

1. Let `payout_input` be the transaction input being spent. Let `payout_datum` be its datum.
2. Let `reserve_input` be the sole transaction input from `reserve_addr`.
3. Let `payout_output` be a transaction output.
4. `payout_input` and `payout_output` must each hold exactly one `payout` token and it must match between them.
5. Let `value_diff` be the sum of values of `payout_input` and `reserve_input`, minus the value of `payout`.
6. `value_diff` must not be negative.
7. If `value_diff` is positive:
    - The transaction must send a change output to `reserve_addr` with a value that matches or exceeds `value_diff`.
8. The transaction must not mint or burn any tokens.

Complete Payout. When the payout accumulator contains sufficient funds, complete the payout to the destination address and datum. Conditions:

1. Let `payout_input` be the transaction input being spent. Let `payout_datum` be its datum.
2. `payout_input` must hold exactly one token of `payout`, which must be the only `payout` burned in the transaction.
3. Let `payout_output` be a transaction output.
4. `payout_output.address` must match `payout_datum.l1_address`.
5. `payout_output.datum` must match `payout_datum.l1_datum`.
6. `payout_output.value` must match or exceed `payout_datum.l2_value`, excluding the burned token.
7. The transaction must not mint or burn any other tokens.

3.8 Midgard hub oracle

This oracle keeps track of minting policy IDs and spending validator addresses for the lists used in the operator directory, state queue, and fraud proof set of the Midgard protocol. It consists of a single utxo holding the hub oracle NFT and a datum of the following type:

$$
\text{HubOracleDatum} := \left\{
\begin{array}{ll}
\text{registered\_operators} : & \text{PolicyId} \\
\text{active\_operators} : & \text{PolicyId} \\
\text{retired\_operators} : & \text{PolicyId} \\
\text{scheduler} : & \text{PolicyId} \\
\text{state\_queue} : & \text{PolicyId} \\
\text{fraud\_proof\_catalogue} : & \text{PolicyId} \\
\text{fraud\_proof} : & \text{PolicyId} \\
\text{deposit} : & \text{PolicyId} \\
\text{withdrawal} : & \text{PolicyId} \\
\text{tx\_order} : & \text{PolicyId} \\
\text{settlement\_queue} : & \text{PolicyId} \\
\text{payout} : & \text{PolicyId} \\
 & \\
\text{registered\_operators\_addr} : & \text{Address} \\
\text{active\_operators\_addr} : & \text{Address} \\
\text{retired\_operators\_addr} : & \text{Address} \\
\text{scheduler\_addr} : & \text{Address} \\
\text{state\_queue\_addr} : & \text{Address} \\
\text{fraud\_proof\_catalogue\_addr} : & \text{Address} \\
\text{fraud\_proof\_addr} : & \text{Address} \\
\text{deposit\_addr} : & \text{Address} \\
\text{withdrawal\_addr} : & \text{Address} \\
\text{tx\_order\_addr} : & \text{Address} \\
\text{settlement\_queue\_addr} : & \text{Address} \\
\text{reserve\_addr} : & \text{Address} \\
\text{payout\_addr} : & \text{Address}
\end{array}
\right\}
$$

Add missing policy IDs and addresses to the hub oracle datum.

TODO

3.8.1 Minting policy

The `hub_oracle` minting policy ensures that all Midgard lists are initialized together and sent to their respective spending validator addresses. Redeemers:

Init. Initialize all Midgard lists and send their root nodes to their respective validator addresses. Conditions:

1. Let `nonce_utxo` be a static parameter of the `hub_oracle` minting policy.
2. `nonce_utxo` must be spent.

3. The hub oracle NFT must be minted.
4. Let `hub_oracle_output` be the transaction output with the hub oracle NFT.
5. `hub_oracle_output` must *not* contain any other non-ADA tokens.
6. `hub_oracle_output` must be sent to the `burn_everything` spending validator address.
7. The root node NFT of every linked list policy ID in `hub_oracle_output` must be minted and sent to the corresponding spending validator address in `hub_oracle`.
8. The NFT of the `scheduler` policy ID in `hub_oracle_output` must be minted and sent to the `scheduler_addr`.
9. No other tokens must be minted or burned.

The nonce utxo proves authority for initialization — whoever controls it is authorized to initialize the Midgard L1 data structures.

3.8.2 Spending validator

The spending validator of `hub_oracle` does not allow its utxo to be spent.

Chapter 4

Proof protocol

4.1 Fraud proof catalogue

For every possible violation of Midgard's ledger rules (see Chapter 5), Midgard defines an onchain procedure (see Section 4.3) to verify whether a given fraud proof demonstrates the existence of that violation in a given block.

Midgard's fraud proof catalogue defines the universe of fraud proof verification procedures. It maps a unique 4-byte integer ($\mathcal{B}_4$) index to the first step of each procedure.[1] On Cardano L1, the Merkle Patricia Trie (MPT) root hash ($\mathcal{RH}$) of this map is stored in a single designated utxo created at Midgard's initialization.

$$\begin{aligned}\mathsf{FraudProofCatalogueDatum} &:= \mathcal{RH}\Big(\mathsf{Map}(\mathcal{B}_4, \mathsf{ScriptHash})\Big) \\ &:= \mathcal{RH}\Big(\Big\{(k_i : \mathcal{B}_4, v_i : \mathsf{ScriptHash}) \mid \forall i \neq j.\ k_i \neq k_j\Big\}\Big)\end{aligned}$$

4.1.1 Minting policy

The `fraud_proof_catalogue` minting policy is statically parametrized on the `hub_oracle` minting policy. Redeemers:

Mint. Initialize the `fraud_proof_catalogue` via the Midgard hub oracle. Conditions:

1. The transaction must mint the Midgard hub oracle NFT.

4.1.2 Spending validator

The spending validator of `fraud_proof_catalogue_addr` does *not* allow its utxo to be spent. Midgard's fraud proof verification procedures are defined once and for all at initialization.

[1] Using 4-byte integer indices means that the catalogue can store up to 4096 procedures, which is more than sufficient for Midgard's ledger rules.

4.2 Fraud proof tokens

Fraud proof tokens represent fraud proof computations that have successfully concluded.

4.2.1 Minting policy

The `fraud_proof` minting policy is statically parametrized on the `computation_thread` and `hub_oracle` minting policies. Redeemers:

Mint. Mint a new fraud proof token whenever a fraud proof computation succeeds. Conditions:

1. The transaction must burn a `computation_thread` token.
2. The transaction must mint a `fraud_proof` token with the same token name as the `computation_thread` token.
3. The transaction must include the Midgard hub oracle NFT in a reference input.
4. Let `fraud_proof_addr` be the policy ID in the corresponding field of the Midgard hub oracle.
5. The `fraud_proof` token must be sent to the `fraud_proof_addr` spending validator.

4.2.2 Spending validator

The spending validator of `fraud_proof_addr` does *not* allow its utxo to be spent. Midgard fraud proofs last forever.

4.3 Fraud proof computation threads

A computation thread splits up a large computation into a series of steps, passing control between the steps in the continuation-passing style (CPS). In other words, the computation thread is a state machine (see Section A.2) with a linear state graph.

Each step is a spending validator that executes the following sequence:

1. Parses the computation state from its datum.
2. Optionally receives arguments from its redeemer to guide the computation step.
3. Advances the computation by the step, producing a new computation state.
4. Suspends the computation by serializing the new computation state into a new datum that it sends to the spending validator of the next step.

All steps in a computation thread parametrize the same datum type:

$$\text{StepDatum}(\text{state\_data}) := \left\{ \begin{array}{ll} \text{fraud\_prover} : & \text{PubKeyHash} \\ \text{data} : & \text{state\_data} \end{array} \right\}$$

4.3.1 Minting policy

The `computation_thread` minting policy initializes its state machine. It is statically parametrized on the `fraud_proof_catalogue` and `hub_oracle` minting policies. Redeemers:

Init. Conditions:

1. The transaction must include the `fraud_proof_catalogue` NFT in a reference input.
2. Let `fraud_category_id` and `fraud_category` be a four-byte key and its corresponding value (a script hash) in the MPT root hash stored in the `fraud_proof_catalogue`.
3. The transaction must include the Midgard hub oracle NFT in a reference input.
4. Let `state_queue` be the policy ID in the corresponding field of the Midgard hub oracle.
5. The transaction must reference a `state_queue` node. Let `fraud_node` be that node.
6. The transaction must mint a single token of the `computation_thread` minting policy. The token name must concatenate `fraud_category_id` and the 28-byte block header hash in the key of `fraud_node`.
7. The computation thread token must be sent to the spending validator address of `fraud_category`. Let `output_state` be that transaction output.
8. The `output_state` datum type must be `StepDatum(Void)`.[2]
9. The transaction must be signed by the `fraud_prover` pub-key hash of `output_state`.
10. Other than ADA, `output_state` must *not* hold any other tokens.
11. The transaction must *not* mint or burn any other tokens.

[2] Aiken calls it StepDatum(Void), while Plutarch calls it StepDatum(PUnit).

Success. Terminate the state machine normally from the final spending validator in the computation. There are no conditions because this redeemer relies on the spending validator to burn the token.

Cancel. Terminate the state machine exceptionally from any spending validator in the computation. There are no conditions because this redeemer relies on the spending validator to burn the token.

4.3.2 Spending validators

A fraud-proof computation succeeds if its thread token passes through all the steps' spending validator addresses. In that case, the last step's spending validator reifies the successful fraud-proof by requiring a fraud-proof token to be minted. The fraud-proof minting policy requires the computation thread token to be burned, specifically via the Success redeemer.

On the other hand, at any step, the person who initiated the computation thread can cancel the computation instead of advancing it. In that case, the step's spending validator requires the computation thread token to be burned via the Cancel redeemer. Thus, while there is typically only one path for a computation thread to reach success via the sequential steps,[3] there may be multiple opportunities for the computation to be canceled along the way.

For each fraud-proof category, each spending validator is custom-written to express the specific logic of that computation step, and it is statically parametrized on the next step's spending validator (if any). One of the custom conditions of the computation step should verify the transition between the input state and output state of the thread:

$$\begin{aligned} \text{verify\_transition} &: (\text{Input}, \text{Output}, ..\text{Args}) \to \text{Bool} \\ \text{verify\_transition}(\text{i}, \text{o}, ..\text{args}) &:= \Big(\text{transition}(\text{i}, ..\text{args}) \equiv \text{o}\Big) \\ \text{transition} &: (\text{Input}, ..\text{Args}) \to \text{Output} \end{aligned}$$

All of the spending validators share the same parametric redeemer type, but each spending validator can parametrize the Continue redeemer by a different type to hold the custom instructions needed to guide the computation step:

$$\text{StepRedeemer}(\text{instructions}) := \text{Continue}(\text{instructions}) \mid \text{Cancel}$$

These redeemers should be handled in the following general pattern:

Continue. Advance the computation. Conditions:

1. If this is the last step of the computation:
    - Mint the fraud token, which will implicitly burn the computation thread token with the Success redeemer. Let `output_state` be that transaction output.
    - The `output_state` datum type must be `StepDatum(Void)`.

[3] Technically, if there are multiple instances of the same fraud category in a fraudulent block, then there is a corresponding number of paths to prove the occurrence of that fraud category in the block. The redeemer arguments provided to the computation steps collectively select one of these paths.

- The `fraud_prover` field must match between the `output_state` and the input datum.

2. Otherwise:

   - The computation thread token must be sent to the next step's spending validator. Let `output_state` be that transaction output.

   - The `fraud_prover` field must match between the `output_state` and the input datum.

3. Evaluate the custom conditions of the computation step, including verifying the state transition.

4. The custom conditions may require the transaction to reference a state queue with a key hash matching the last 28 bytes of the computation thread token name.

5. The transaction must *not* mint or burn any other tokens.

Cancel. Cancel the computation. Conditions:

1. Burn the computation thread token with the Cancel redeemer.

2. Return the ADA from the computation thread utxo to the fraud prover pub-key defined in the input datum.

3. The transaction must *not* mint or burn any other tokens.

Chapter 5

Ledger rules and fraud proofs

5.1 Midgard Ledger Rules and Fraud Proofs

In the following sections the following premises are used:

$$
\begin{aligned}
b &\in Blocks \\
txs &:= transactions(b) \\
utxos_{pre} &:= prev\_utxos(b) \\
utxos_{post} &:= utxos(b) \\
wtxs &:= withdrawals(b)
\end{aligned}
$$

5.1.1 Rule: All inputs must be valid

A transaction cannot spend a non-existing (or an already spent) UTxO. Formal specification:

$$
\begin{aligned}
&\forall t \in Ledger,\ \forall i \in spend\_inputs(t) : \\
&\quad (\exists t_1 \in Ledger,\ t \neq t_1\ \wedge\ i \in outputs(t_1)) \wedge \\
&\quad (\nexists t_2 \in Ledger,\ t \neq t_2\ \wedge\ i \in spend\_inputs(t_2))
\end{aligned}
$$

This ledger rule is violated if any of the following violations occur:

- NO-INPUT
- INPUT-NO-IDX
- WITHDRAWN-INPUT
- DOUBLE-SPEND
- DOUBLE-WITHDRAW

NO-INPUT violation

A transaction t attempted to spend the UTxO i that does not exist or was spent in a previous block. Formal specification:

$$
\begin{aligned}
&\exists t \in txs,\ \exists i \in spend\_inputs(t) : \\
&\quad (i \notin utxos_{prev}) \wedge \\
&\quad (\nexists t_1 \in txs,\ t \neq t_1\ \wedge\ tx\_hash(t_1) = tx\_hash(i))
\end{aligned}
$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for t that shows that the transaction is included in the block ($t \in txs$).
3. A membership proof for input i must be specified such that $i \in spend\_inputs(t)$.
4. A non-membership proof must be created to show that i is not in $utxos_{prev}$.
5. A non-membership proof must also be generated that shows that there are no transactions in txs that have id $tx\_hash(i)$.

INPUT-NO-IDX violation

A transaction t attempted to spend the input i, which did not exist at all. Formal specification:

$$\exists t \in txs,\ \exists i \in spend\_inputs(t),\ \exists t_1 \in txs :$$
$$tx\_hash(t_1) = tx\_hash(i)\ \wedge\ i \notin outputs(t_1)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for t that shows that the transaction is included in the block ($t \in txs$)
3. A membership proof must be presented for input i such that $i \in spend\_inputs(t)$
4. A membership proof must be created for t_1 such that $tx\_hash(t_1) = tx\_hash(i)$
5. A DA layer proof must be presented that certifies that $length(outputs(t_1)) < index(i)$

WITHDRAWN-INPUT violation

A transaction t attempted to spend the input i, which was spent in a withdraw transaction. Formal specification:

$$\exists t \in txs,\ \exists i \in spend\_inputs(t),\ \exists w \in wtxs :$$
$$i = l2\_outref(w)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for t that shows that the transaction is included in the block ($t \in txs$)
3. A membership proof must be generated for input i such that $i \in spend\_inputs(t)$
4. A membership proof must be created to show that w is in $wtxs$, which also spends input i

DOUBLE-SPEND violation

A transaction t attempted to spend the input i, which was spent in another transaction. Formal specification:

$$\exists t \in txs,\ \exists i \in spend\_inputs(t),\ \exists t_1 \in tx :$$
$$t \neq t_1\ \wedge\ i \in spend\_inputs(t_1)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for t that shows that the transaction is included in the block ($t \in txs$)
3. A membership proof must be generated for input i such that $i \in spend\_inputs(t)$
4. A membership proof must be created to show that t_1 is in txs
5. A membership proof must be given that verifies that $i \in spend\_inputs(t_1)$

DOUBLE-WITHDRAW violation

A withdrawal w attempted to spend the same input, which was already withdrawn by w_1. Formal specification:

$$\exists w,\ w_1 \in wtxs :$$
$$w \neq w_1\ \wedge\ l2\_outref(w) = l2\_outref(w_1)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for w that shows that the withdraw transaction is included in the block ($w \in wtxs$)
3. A membership proof must be created to show that w_1 is in $wtxs$

5.1.2 Rule: Transaction validity range

Every valid transaction in the ledger must be included at a timestamp that conforms to the validity range that the transaction prescribes. Formal specification:

$$\forall t \in Ledger :$$
$$time\_range(block(t)) \subseteq validity\_interval(t)$$

This ledger rule is violated if any of the following violations occur:

- INVALID-RANGE

INVALID-RANGE violation

A transaction t has a time-validity range that does not overlap with its block's event interval. Formal specification:

$$\exists t \in txs :$$
$$time\_range(b) \not\subseteq validity\_interval(t)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that proves that transaction t is included in the block b

5.1.3 Rule: At least one input

Every valid transaction in the ledger must spend at least one UTxO. Formal specification:

$$\forall t \in Ledger :$$
$$|spend\_inputs(t)| > 0$$

This ledger rule is violated if any of the following violations occur:

- ZERO-INPUT

ZERO-INPUT violation

A transaction t is in the ledger and spends no inputs. Formal specification:

$$\exists t \in txs :$$
$$|spend\_inputs(t)| = 0$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction is included in the ledger
3. A DA layer proof must be presented that certifies that $length(spend\_inputs(t)) = 0$

5.1.4 Rule: Minimum fee

Every valid transaction in the ledger must pay the fees for inclusion. Formal specification:

$$\forall t \in Ledger :$$
$$tx\_fee(t) \geq min\_fee(t)$$

The fee calculation algorithm is the same as in Cardano ... **TODO**

This ledger rule is violated if any of the following violations occur:

- MIN-FEE

MIN-FEE violation

A transaction t is in the ledger, while $fee(t) < min\_fee(t)$. Formal specification:

$$\begin{aligned} &\exists t \in txs : \\ &\quad fee(t) < min\_fee(t) \end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction is included in the ledger

5.1.5 Rule: Required signatures are correct

Every valid transaction in the ledger must correctly show the required signers. Formal specification:

$$\forall t \in Ledger : \\ required\_signer\_hashes(t) = \left\{ paymentHK(addr(u)) \;\middle|\; \begin{array}{l} (r,u) \in utxos \\ r \in spend\_inputs(t) \\ addr(u) \in Addr^{vkey} \end{array} \right\}$$

This ledger rule is violated if any of the following violations occur:

- MISSING-REQ-SIGNER-TX
- MISSING-REQ-SIGNER-UTXO
- NON-REQ-SIGNER

MISSING-REQ-SIGNER-TX violation

A transaction t_1 spends a utxo u, produced in the block by transaction t_2 at a public key address, without providing the required signature as a witness. Formal specification:

$$\begin{aligned} &\exists t_1, t_2 \in txs,\ \exists i \in spend\_inputs(t_1)\ \exists u \in outputs(t_2) : \\ &(tx\_hash(i) = tx\_hash(t_2)) \wedge \\ &(u = elem\_at(outputs(t_2), i.index)) \wedge \\ &\quad paymentHK(addr(u)) \notin required\_signer\_hashes(t_1) \wedge \\ &\quad addr(u) \in Addr^{vkey} \end{aligned}$$

Fraud proof construction:

1. Let t_1 be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that t_1 is included in the block.
3. Let t_2 be another transaction in the block.

4. A membership proof must be provided that shows that t_2 is included in the block.
5. Verify that $i \in spend\_inputs(t_1)$.
6. Verify that i matches t_2 on transaction hash.
7. Let u be the t_2 output indexed by i.
8. Verify that the payment credential of u is a public key hash and that it is included in the required signers of the transaction.

MISSING-REQ-SIGNER-UTXO violation

A transaction t_1 spends a utxo u from the previous block's utxo set without providing the required signature as a witness. Formal specification:

$$\begin{aligned}&\exists t_1 \in txs,\ \exists i \in spend\_inputs(t_1)\ \exists (j,u) \in utxos_{pre} :\\ &(i = j) \wedge\\ &\quad paymentHK(addr(u)) \notin required\_signer\_hashes(t_1) \wedge\\ &\quad addr(u) \in Addr^{vkey}\end{aligned}$$

Fraud proof construction:

1. Let t_1 be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that t_1 is included in the block.
3. Verify that $i \in spend\_inputs(t_1)$.
4. Verify that i matches t_2 on transaction hash.
5. Let u be a utxo and j be an output reference.
6. A membership proof must be provided that $(j,u) \in utxos_{pre}$.
7. Verify that the payment credential of u is a public key hash and that it is included in the required signers of the transaction.

NON-REQ-SIGNER violation

A transaction t is in the ledger, and it violates the "Required signatures" property. Formal specification:

$$\begin{aligned}&\exists t \in txs,\ \exists vkey \in required\_signer\_hashes(t),\ \nexists (r,u) \in utxos_{post} :\\ &\quad r \in spend\_inputs(t) \wedge\\ &\quad paymentHK(addr(u)) = vkey \wedge\\ &\quad addr(u) \in Addr^{vkey}\end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.

2. A membership proof must be provided that shows that the transaction t is included in the ledger
3. A DA layer proof must be presented that shows that for a specified $vkey$ there is no utxo $u \in utxos_{post}$, such that the address of u corresponds to $vkey$ and t spends u

5.1.6 Rule: Signatures are valid

Every provided signature must be valid. Formal specification:

$$\forall t \in Ledger,\ \forall (v, s, h) \in addr\_tx\_wits(t) :$$
$$is\_valid\_signature(v, s, h)$$

This ledger rule is violated if any of the following violations occur:

- INVALID-SIGNATURE

INVALID-SIGNATURE violation

There exists an invalid signature for transaction t (if indeed the signature (v, s, h) is not valid). Formal specification:

$$\exists t \in txs,\ \exists (v, s, h) \in addr\_tx\_wits(t) :$$
$$\neg is\_valid\_signature(v, s, h)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger
3. A membership proof must be shown that states that $(v, s, h) \in addr\_tx\_wits(t)$

5.1.7 Rule: Every needed signature is provided

Every required signature is provided. Formal specification:

$$\forall t \in Ledger,\ \forall h \in required\_signer\_hashes(t) :$$
$$\exists (v, s, h) \in addr\_tx\_wits(t)$$

This ledger rule is violated if any of the following violations occur:

- MISSING-SIGNATURE

MISSING-SIGNATURE violation

A required signature, corresponding to h is missing. Formal specification:

$$\exists t \in txs,\ \exists h \in required\_signer\_hashes(t) :$$
$$h \notin \{h_p \mid (v, s, h_p) \in addr\_tx\_wits(t)\}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger
3. A DA layer proof must be shown that states that $h \in required\_signer\_hashes(t)$
4. A DA layer proof must be presented that shows that a signature with h does not exist

5.1.8 Rule: Native scripts are available

All native scripts are provided. Formal specification:

$$\forall t \in Ledger,\ \forall u \in output(t) :$$
$$addr(u) \in Addr_{v2}^{native} \implies \left(\exists (h, s) \in script\_tx\_wits(t),\ script\_hash(addr(u)) = h\right)$$

This ledger rule is violated if any of the following violations occur:

- MISSING-NATIVE-SCRIPT

MISSING-NATIVE-SCRIPT violation

A required script, corresponding to h is missing. Formal specification:

$$\exists t \in txs,\ \exists u \in output(t) :$$
$$addr(u) \in Addr_{v2}^{native} \wedge$$
$$\left(\nexists (h, s) \in script\_tx\_wits(t),\ script\_hash(addr(u)) = h\right)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger.
3. A membership proof must be generated that proves that u is in $outputs(t)$.
4. A DA layer proof must be shown that certifies that there is no (h, s) corresponding to $script\_hash(addr(u)) = h$ in $script\_tx\_wits(t)$.

5.1.9 Rule: Native scripts validated

All native scripts validations pass. Formal specification:

$$\forall t \in Ledger,\ \forall (h, s) \in script\_tx\_wits(t) :$$
$$native\_validation\_succeeds(s, t)$$

TODO

This ledger rule is violated if any of the following violations occur:

- NATIVE-SCRIPT-INVALID

NATIVE-SCRIPT-INVALID violation

A native script s validation fails. Formal specification:

$$\exists t \in txs,\ \exists (h, s) \in script\_tx\_wits(t) :$$
$$\neg native\_validation\_succeeds(s, t)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger.
3. A DA layer proof must be shown that certifies that (h, s) exists in $script\_tx\_wits(t)$.
4. The $native\_validation\_succeeds(s, t)$ fails.

5.1.10 Rule: Value preservation

The total value must be preserved. Formal specification:

$$\forall t \in Ledger :$$
$$mint(t) + \sum value(spend\_inputs(t)) = fee(t) + \sum value(outputs(t))$$

This ledger rule is violated if any of the following violations occur:

- VALUE-NOT-PRESERVED

VALUE-NOT-PRESERVED violation

Value is not preserved in transaction t. Formal specification:

$$\exists t \in txs :$$
$$mint(t) + \sum value(spend\_inputs(t)) \neq fee(t) + \sum value(outputs(t))$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger.
3. A DA layer proof must be shown that certifies the value of $sum(spend\_inputs(t))$.
4. A DA layer proof must be created that states the value of $sum(outputs(t))$.
5. The sums in the calculation must show a discrepancy.

5.1.11 Rule: No Ada minted

Ada must not be minted. Formal specification:

$$\begin{aligned}&\forall t \in Ledger :\\ &\quad lovelaces(mint(t)) = 0\end{aligned}$$

This ledger rule is violated if any of the following violations occur:

- ADA-MINTED

ADA-MINTED violation

Ada is minted in transaction t. Formal specification:

$$\begin{aligned}&\exists t \in txs :\\ &\quad lovelaces(mint(t)) \neq 0\end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger.
3. $lovelaces(mint(t)) \neq 0$

5.1.12 Rule: No negative value

All output values must be greater or equal to zero. Formal specification:

$$\begin{aligned}&\forall t \in Ledger :\\ &\quad \left(\forall o \in outputs(t),\ value(o) \geq \mathbf{0}\right)\end{aligned}$$

This ledger rule is violated if any of the following violations occur:

- NEGATIVE-OUTPUT-VALUE

NEGATIVE-OUTPUT-VALUE violation

Ada is minted in transaction t. Formal specification:

$$\exists t \in txs,\ \exists o \in outputs(t),\ \exists m \in Policy,\ \exists tn \in TokenName :$$
$$value(o)_{m,tn} < 0$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that the transaction t is included in the ledger.
3. A membership proof must be generated that states that $o \in outputs(t)$.
4. A minting policy id m and a token name tn must be shown, such that the quantity of (m, tn) tokens in o is negative.

5.1.13 Rule: Minimum UTxO value

All output values must adhere to the same minimum ada value requirements that exists on Cardano. Formal specification:

$$\forall t \in Ledger,\ \forall o \in outputs(t) :$$
$$lovelaces(value(o)) \geq min\_ada\_value(o)$$

TODO

This ledger rule is violated if any of the following violations occur:

- MIN-ADA-TX
- MIN-ADA-UTXO

MIN-ADA-TX violation

An output of a transaction t does not satisfy the minimum Ada value requirement. Formal specification:

$$\exists t \in txs,\ \exists o \in outputs(t) :$$
$$lovelaces(value(o)) < min\_ada\_value(o)$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that $t \in txs$.
3. A membership proof must be provided that shows that $o \in outputs(t)$.
4. $lovelaces(value(o)) < min\_ada\_value(o)$ must hold.

MIN-ADA-UTXO violation

A utxo u in a block's utxo set does not satisfy the minimum Ada value requirement. Formal specification:

$$\begin{aligned}&\exists u \in utxos_{post} : \\ &\quad lovelaces(value(u)) < min\_ada\_value(u)\end{aligned}$$

Fraud proof construction:

1. Let u be the utxo alleged to violate the ledger rule.
2. A membership proof must be provided that shows that $u \in utxos_{post}$.
3. $lovelaces(value(u)) < min\_ada\_value(u)$ must hold.

5.1.14 Rule: Network id of outputs

All output addresses must have the correct network id. Formal specification:

$$\begin{aligned}&\forall t \in Ledger,\ \forall o \in outputs(t) : \\ &\quad network\_id(addr(o)) = network\_id_{Midgard}\end{aligned}$$

This ledger rule is violated if any of the following violations occur:

- OUTPUT-NETWORK-UTXO
- OUTPUT-NETWORK-TX

OUTPUT-NETWORK-TX violation

An output of a transaction t has a wrong network id. Formal specification:

$$\begin{aligned}&\exists t \in txs,\ \exists o \in outputs(t) : \\ &\quad network\_id(addr(o)) \neq network\_id_{Midgard}\end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that $t \in txs$.
3. A membership proof must be provided that $o \in outputs(t)$.
4. $network\_id(addr(o)) \neq network\_id_{Midgard}$ must hold.

OUTPUT-NETWORK-UTXO violation

A UTxO u address has a wrong network id. Formal specification:

$$\begin{aligned}&\exists u \in utxos_{post} : \\ &\quad network\_id(addr(o)) \neq network\_id_{Midgard}\end{aligned}$$

Fraud proof construction:

1. Let u be the output alleged to violate the ledger rule.
2. A membership proof must be provided that shows that $u \in utxos_{post}$.
3. $network\_id(addr(o)) \neq network\_id_{Midgard}$ must hold.

5.1.15 Rule: Network id of transaction

All transactions must have the correct network id. Formal specification:

$$\begin{aligned}&\forall t \in Ledger : \\ &\quad network\_id(t) = network\_id_{Midgard}\end{aligned}$$

This ledger rule is violated if any of the following violations occur:

- TRANSACTION-NETWORK

TRANSACTION-NETWORK violation

A transaction t has a wrong network ID. Formal specification:

$$\begin{aligned}&\exists t \in txs : \\ &\quad network\_id(t) \neq network\_id_{Midgard}\end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided that shows that $t \in txs$.
3. $network\_id(t) \neq network\_id_{Midgard}$ must hold.

5.1.16 Rule: All reference inputs must be valid

A transaction can only reference utxos that were produced by transactions that do not follow the transaction in the ledger's order, and it cannot reference utxos that were spent in preceding transactions. Formal specification:

$$\begin{aligned}&\forall t \in Ledger,\ \forall r \in reference\_inputs(t) : \\ &\quad (\exists t_1 \in Ledger,\ t_1 \preccurlyeq t\ \wedge\ r \in outputs(t_1)) \wedge \\ &\quad (\nexists t_2 \in Ledger,\ t_2 \prec t\ \wedge\ r \in spent\_inputs(t_2))\end{aligned}$$

This ledger rule is violated if any of the following violations occur:

- NO-REFERENCE-INPUT
- REFERENCE-INPUT-NO-IDX

NO-REFERENCE-INPUT violation

A transaction t attempted to spend the UTxO i that does not exist or was spent in a previous block. Formal specification:

$$\begin{aligned}&\exists t \in txs,\ \exists i \in reference\_inputs(t) : \\ &\quad (i \notin utxos_{prev}) \land \\ &\quad (\nexists t_1 \in txs,\ t \neq t_1 \land tx\_hash(t_1) = tx\_hash(i))\end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for t that shows that $t \in txs$.
3. A membership proof for input i must be provided such that $i \in reference\_inputs(t)$.
4. A non-membership proof must be provided to show that $i \notin utxos_{prev}$.
5. A non-membership proof must also be generated that shows that no transactions in the block match the input's tx hash.

REFERENCE-INPUT-NO-IDX violation

A transaction t attempted to spend an input i that was not produced by the transaction matching the tx hash of i. Formal specification:

$$\begin{aligned}&\exists t \in txs,\ \exists i \in reference\_inputs(t) : \\ &\quad (\exists t_1 \in txs,\ tx\_hash(t_1) = tx\_hash(i)\ \land\ length(outputs(t_1) < index(i))\end{aligned}$$

Fraud proof construction:

1. Let t be the transaction alleged to violate the ledger rule.
2. A membership proof must be provided for t that shows that the transaction is included in the block ($t \in txs$)
3. A membership proof must be presented for input i such that $i \in reference\_inputs(t)$
4. A membership proof must be created for t_1 such that $tx\_hash(t_1) = tx\_hash(i)$
5. A DA layer proof must be presented that certifies that $length(outputs(t_1)) < index(i)$

5.2 Custom Midgard ledger rules

5.2.1 Rule: Is valid

5.2.2 Rule: No auxiliary data

5.2.3 Rule: Certificates empty

5.2.4 Rule: Only zero withdrawals

5.2.5 Rule: No mints

5.3 Data availability rules

5.3.1 Rule: Transaction count correct

Chapter 6

Offchain data architecture

Midgard operators only commit block headers to the L1 state queue. They store their actual blocks temporarily in Midgard's data availability layer for at least the maturity period, and then permanently on Midgard's archive nodes.

6.1 Data availability layer

The data availability layer is critical to Midgard's security because every committed block needs to be publicly available throughout the maturity period so that watchers can detect and prove fraud before invalid blocks are merged.

We are considering three solutions for the data availability layer (in decreasing preference):

1. Cardano Leios blobs (the ideal solution)
2. Multi-signature committees

6.1.1 Data availability via Leios blobs

The ideal data availability solution for Midgard is based on Cardano Leios blobs, which are a proposed feature in Cardano that will support large-scale transient data storage secured via L1 consensus.

The contents of these blobs do not have to be verified by Cardano's Ouroboros consensus protocol, and they only have to be stored for up to 30 days. This means that a large amount of data can be stored in these blobs sustainably for a low cost, which we expect to be several orders of magnitude lower than Cardano's cost for transaction metadata (which is stored permanently).

Leios blobs are a natural intermediate point on Leios' multi-year roadmap toward full input-endorser capabilities. We believe that they are achievable before Midgard's planned deployment on mainnet, and we will support the Leios team to help bring them to Cardano sooner.

In the Leios-based data availability solution for Midgard, operators will pay to store their full non-Merkelized blocks inside Leios blobs for the full maturity period. Leios itself will provide timestamps and (non-Merkle) hashes for the blobs and ensure that the blob contents are accessible. Midgard's L1 smart contracts will be able to access the timestamps and non-Merkle hashes of blobs directly.

The block data inside each Leios blob will be sufficient to be converted offchain into the Merkelized representation of Midgard blocks that is necessary to construct fraud proofs. The correspondence between the non-Merkelized block data stored in the Leios blob and the Merkle root hash declared in the block header will be verifiable via a special fraud proof verification procedure. This procedure will calculate the Merkle root hash in a streaming fashion over the block data and compare it to the declared Merkle root hash in the block header.

We hope to further streamline this Merkle root hash verification procedure by collaborating with the Cardano Leios and Plutus teams to make it a Plutus builtin operation. This builtin can be much more expensive than typical Plutus built-ins, and Midgard fraud provers will be happy to pay the cost because the stakes of DA fraud and the reward for proving it are much higher. Nonetheless, this will not impose an ongoing cost burden on Midgard because this verification procedure will not need to be invoked during normal operation with honest blocks, and invalid blocks will be rare.

Operators' costs for storing blocks in Leios will be offset by the revenue they collect from Midgard transaction, deposit, and withdrawal fees. Furthermore, the Leios blob storage fees will become an additional source of revenue for Cardano L1 block producing nodes, further boosting the economic security of Cardano L1 on which Midgard depends.

6.1.2 Data availability via Mithril

If Leios blobs are unavailable on Cardano mainnet in time for Midgard's deployment, a viable alternative is using stake-weighted Mithril certificates to ensure data availability.

The Mithril whitepaper states that it was specifically designed to address the data-availability problem:

> "Mithril provides an immediate solution to the data-availability problem: if the underlying consensus protocol is run on references for which a Mithril signature exists, data availability is (cryptographically) guaranteed."

In this approach, a state commitment is only considered valid if it is accompanied by a Mithril certificate signed by a stake-weighted quorum of Mithril participants. This certificate represents the Mithril participants' collective claim that the necessary block data is publicly available.

Midgard's L1 consensus protocol is adapted as follows:

1. **Publishing Data:** When the current operator wishes to commit a new block header to the state queue, they must publish the block's full data at a publicly accessible location. This data must be sufficient to reconstruct the Merkle roots referenced by the block header.
2. **Verification by Mithril Participants:** Each Mithril participant monitors the publicly accessible location. When a new block is published, the Mithril participant independently retrieves the block's data and verifies that:
    - The data is available.
    - It corresponds exactly to the Merkle roots in the state commitment.

3. **Signing the Certificate:** If the block's data meets these conditions, the Mithril participant signs the Mithril certificate and broadcasts the signature to the Mithril aggregators.

4. **Onchain Verification:** The Midgard state queue requires every new block header to be appended with an associated Mithril certificate signed by Midgard's quorum for the Mithril-based DA. Midgard's `da_multisig_threshold` parameter sets this quorum threshold.

By leveraging Mithril, Midgard ensures that state commitments are only accepted when their corresponding block data is provably available. This mechanism safeguards Midgard against data-availability fraud, where a malicious operator attempts to submit a state commitment without disclosing the underlying data. Without access to this data, fraud provers would be unable to construct and submit fraud proofs, allowing the fraudulent state commitment to become canonical once the fraud detection window closes. By requiring a Mithril certificate before a state commitment is appended, Midgard guarantees that the data remains accessible, preserving the system's integrity.

A key difference between the Leios blob and Mithril-based data availability (DA) solutions lies in the security guarantees each approach provides. In the Leios blob solution, data availability is assured with the full security of Cardano's consensus, meaning that all active stake on the network contributes to securing DA, just as it does for transaction finality and block production. Thus, any attempt to suppress or manipulate stored data would require an adversary to control a majority of Cardano's total stake, making attacks highly impractical. In contrast, the Mithril-based DA solution relies on a subset of Cardano's stake—specifically, the stake controlled by SPOs (Stake Pool Operators) running Mithril or, equivalently, the total stake actively participating in Mithril. While this still provides strong cryptographic guarantees, it does not benefit from the full economic security of Cardano's entire stake-weighted consensus, making it theoretically more susceptible to adversarial control if a sufficiently large fraction of the Mithril-active stake colludes or becomes compromised.

This difference in security between the Leios blob and Mithril-based data availability (DA) solutions is alleviated as more SPOs participate in Mithril. As the number of SPOs running Mithril nodes grows, an increasing proportion of Cardano's total stake becomes actively involved in securing the DA, thereby increasing the overall economic security of the Mithril-based approach. As Mithril approaches universal adoption among Cardano's SPOs, the Mithril-based DA solution's resilience converges to that of the Leios blob-based DA solution. At the limit, they are equivalent because the same stake-weighted economic security backs both.

Therefore, broad participation in Mithril among Cardano SPOs is key to minimizing any security gap between the two approaches.

6.2 Archive node

Midgard's archive nodes will store the block data for confirmed blocks. Security properties are easier to achieve for this historical data because, at all times, Midgard's confirmed state utxo contains a chained header hash that pins the entire historical chain of confirmed blocks. There can be no disagreement about different versions of this data—an archive node either stores data that corresponds to the confirmed state header hash, or it does not.

We expect all operators to keep a copy of the latest confirmed block's utxo set because it is necessary for them to create honest blocks and avoid being slashed for fraudulent blocks. They are incentivized to do so by the fee revenue they earn on Midgard. This utxo set and the block data stored in the data

availability layer provide Midgard with the minimum information necessary to continue producing blocks.

We expect indexers and other professional service providers to store copies of the Midgard historical data. Midgard's security and ongoing block-production capability does not depend on this historical data. However, we expect it to be stored to the extent that users are willing to pay for this service.

Chapter 7

Phase Two Validation

Phase two validation represents the critical stage in Midgard's transaction processing where UPLC script evaluation occurs.

7.1 Overview

Phase two validation serves several key purposes:

- Validates the execution of UPLC scripts attached to transactions
- Ensures computational bounds are respected
- Maintains verifiable execution traces for fraud proof construction
- Enables efficient dispute resolution through progressive state hashing

The validation process consists of three main components:

1. State representation and management
2. Off-chain script decoding and preparation
3. Execution validation and fraud proof mechanisms

Each component is designed to maintain verifiability while optimizing for L2 performance constraints. The following sections detail these components and their interactions within the Midgard protocol.

7.1.1 Validation Requirements

For a transaction to pass phase two validation:

- All scripts must be successfully decoded from their byte representation
- Script execution must complete within specified resource bounds
- All state transitions must be cryptographically verifiable
- Execution traces must enable efficient fraud proof construction

Branch: (None) @ 8799ca9 • Commit date: 2025-10-22

7.2 State Representation

This section describes how the state is represented during phase two validation, including the data structures and encoding methods used for UPLC evaluation.

7.2.1 Term Representation

The UPLC Term representation uses progressive hashing to maintain compact state while preserving verifiability:

$$\begin{aligned}\text{Term} := \ & \text{Variable}(\text{Name}) \\ & \mid \text{Lambda}(\text{Name}, \text{Hash}) \\ & \mid \text{Apply}(\text{Hash}, \text{Hash}) \\ & \mid \text{Constant}(\text{Hash}) \\ & \mid \text{Force}(\text{Hash}) \\ & \mid \text{Delay}(\text{Hash}) \\ & \mid \text{Builtin}(\text{BuiltinFunction})\end{aligned}$$

Each Hash in the Term representation refers to another Term that has been previously processed and hashed. This creates a directed acyclic graph (DAG) of Term components where larger structures are decomposed into their constituent parts.

For example, a lambda expression like $\lambda x.\lambda y.[yx]$ would be represented as:

- A Lambda node containing (x, h_1)
- Where h_1 is the hash of a Lambda node containing (y, h_2)
- Where h_2 is the hash of an Apply node containing (h_3, h_4)
- Where h_3 is the hash of a Variable node containing y
- Where h_4 is the hash of a Variable node containing x

7.2.2 Decoding Steps

The conversion from flat-encoded script bytes to the final Term follows a sequence of BytesToTermSteps:

$$\text{BytesToTermStep} := \left\{ \begin{array}{ll} \text{remaining\_bytes} : & \text{ScriptBytes} \\ \text{partial\_term} : & \text{Term} \end{array} \right\}$$

Each step represents an atomic transformation in the decoding process, with the decoding_step_proof enabling independent verification of that specific step.

7.2.3 Execution Steps

The execution state during UPLC evaluation is represented by the CEK machine state:

$$\text{CEKState} := \left\{ \begin{array}{ll} \text{term\_hash} : & \text{Hash} \\ \text{env} : & \text{Environment} \\ \text{continuation} : & \text{Continuation} \end{array} \right\}$$

Each execution step produces a new CEK state and tracks execution units consumed:

$$\text{ExecutionStep} := \left\{ \begin{array}{ll} \text{before\_state} : & \text{CEKState} \\ \text{after\_state} : & \text{CEKState} \\ \text{execution\_units} : & \text{ExecutionUnits} \end{array} \right\}$$

7.2.4 Execution Trace

The complete execution trace combines the bytes-to-term conversion and CEK machine evaluation. In the following, we use the notation $\mathcal{RH}$ to indicate a root hash of a Merkle-Patricia tree.

$$\text{ExecutionTrace} := \left\{ \begin{array}{ll} \text{bytes\_to\_term\_steps} : & \mathcal{RH}(\text{[BytesToTermStep]}) \\ \text{initial\_state} : & \text{CEKState} \\ \text{steps} : & \mathcal{RH}(\text{[ExecutionStep]}) \end{array} \right\}$$

The execution trace is stored in transaction witness sets:

$$\text{MidgardTxWits} := \left\{ \begin{array}{ll} \ldots & \\ \text{execution\_traces} : & ?\ \mathcal{RH}(\text{Map(RdmrPtr, ExecutionTrace)}) \end{array} \right\}$$

This representation allows any step of the trace to be independently verified without requiring access to the complete execution history, enabling efficient fraud proof construction and validation.

7.3 Off-chain Decoding

This section details the process of decoding UPLC script bytes into executable Terms during phase two validation.

7.3.1 Byte Format Specification

UPLC scripts are serialized using the flat format, which provides a compact binary representation optimized for blockchain storage. The format consists of:

- Version number (major.minor.patch) encoded as three natural numbers
- Term structure encoded as tagged bit sequences
- Constants encoded based on their type (integers, bytestrings, etc.)
- Built-in functions encoded as 7-bit tags
- De Bruijn indices for variable references
- Padding bits to ensure byte alignment

7.3.2 Decoding Process

The decoding process follows these key steps:

1. Read and validate the version number
2. Parse the term structure by interpreting tag bits:
    - 0011 - Application
    - 0100 - Constants
    - 0111 - Built-in functions
    - etc.
3. Decode constants according to their type tags
4. Resolve built-in function references via their 7-bit tags
5. Construct the final term tree

7.3.3 Term Construction Process

The decoder maintains state during term construction:

$$\text{BytesToTermStep} \coloneqq \left\{ \begin{array}{ll} \text{step\_type} : & \text{DecodingStepType} \\ \text{input\_bytes} : & \text{ScriptBytes} \\ \text{remaining\_bytes} : & \text{ScriptBytes} \\ \text{partial\_term} : & \text{Term} \\ \text{transformation\_proof} : & \text{Hash} \end{array} \right\}$$

7.3.4 Decoding Step Types

The DecodingStepType represents the specific transformation being performed:

$$\text{DecodingStepType} \coloneqq \left\{ \begin{array}{ll} \text{VersionDecode} : & (\text{major, minor, patch}) \\ \text{TermTagDecode} : & \text{TagBits} \\ \text{TypeTagDecode} : & [\text{TypeTag}] \\ \text{ConstantDecode} : & \text{Type} \times \text{Value} \\ \text{BuiltinDecode} : & \text{BuiltinTag} \\ \text{PaddingValidate} : & \text{PaddingBits} \end{array} \right\}$$

7.3.5 Transformation Proofs

Each step's transformation_proof must demonstrate:

- Valid consumption of input bytes according to the format specification
- Correct interpretation of decoded values
- Proper handling of any padding or alignment requirements
- Maintenance of the well-formed term structure

For example, a ConstantDecode proof must show:

- The type tag list was properly terminated
- The constant value matches its declared type
- Any required padding was correctly handled
- The remaining bytes are properly aligned

7.3.6 Validation Chain

The complete decoding process produces a sequence of BytesToTermSteps that can be independently verified. This enables:

- Detection of malformed script bytes
- Identification of specific decoding failures
- Construction of fraud proofs for invalid transformations
- Verification of the complete decoding process

7.3.7 Transformation Types

Different term types require specific decoding transformations:

- Constants - Decoded based on type tags and specific encoding rules
- Applications - Recursively decode function and argument terms
- Built-ins - Lookup via 7-bit tag table
- Variables - Convert de Bruijn indices to term references

7.3.8 Validation Requirements

The decoder must enforce several validation rules:

- Version compatibility check
- Well-formed term structure
- Valid constant values
- Recognized built-in functions
- Proper scope for de Bruijn indices
- Complete byte consumption (no trailing data)

7.4 Fraud Proofs in UPLC Evaluation

This section details the fraud proofs involved in UPLC evaluation, focusing on single-step verification of state transitions.

7.4.1 Types of Fraud Proofs

The system supports these categories of fraud proofs:

Decoding Fraud Proofs of invalid script byte decoding:

- Invalid byte format
- Size limit violations
- Reference resolution failures

Execution Fraud Proofs of invalid UPLC execution:

- Resource limit violations
- Invalid state transitions
- Incorrect execution results

7.4.2 Single-Step Verification

The fraud proof system verifies individual state transitions:

$$\forall s_1, s_2 \in \text{CEKState} : \text{claimed\_transition}(s_1 \to s_2) \text{ valid} \iff \text{compute\_next\_state}(s_1) = s_2$$

To prove a violation:

- The operator provides the claimed before state (s_1) and after state (s_2)
- The challenger computes the actual next state from s_1
- If the computed state differs from s_2, the transition is invalid
- No additional trace information is required

7.4.3 Proof Data Structure

The fraud proof structure is minimal:

$$\text{FraudProof} := \left\{ \begin{array}{ll} \text{step\_number} : & \mathbb{N} \\ \text{before\_state} : & \text{CEKState} \\ \text{claimed\_after\_state} : & \text{CEKState} \\ \text{actual\_after\_state} : & \text{CEKState} \end{array} \right\}$$

7.4.4 Verification Process

The verification process is straightforward:

1. Verify the before state matches the operator's claim
2. Compute one step from the before state
3. Compare computed result with operator's claimed after state
4. If they differ, the fraud proof is valid

7.4.5 Security Considerations

The single-step verification approach provides several benefits:

- Minimal proof size
- Constant-time verification
- No complex challenge periods needed
- Deterministic outcomes

Security guarantees include:

- No false positives (valid transitions cannot be proven invalid)
- No trace reconstruction required
- Immediate verification of claims
- Protection against computational waste attacks

Appendix A

General onchain data structures and mechanisms

This appendix describes general-purpose data structures and mechanisms that can be used for various applications on Cardano. They are designed to be as simple and flexible as possible, and they do not depend on any specifics of the Midgard protocol.

A.1 Linked list

The linked list is a versatile data structure that can implement sets, queues, key-value maps, lazy data sequences, and other interesting data structures. It is particularly useful in the extended UTXO model of the Cardano blockchain, where many list operations and queries can be validated onchain within minimal local contexts that do not grow with list size. Meanwhile, offchain queries can access any part of the list via beacon NFTs that pinpoint specific list nodes.

We describe two types of linked lists:

- The **key-unordered** linked list is a cheaper/simpler variant that only allows nodes to be appended at the end of the list and does not enforce the order of keys in the list. It can be used to implement queues and other sequential data structures. However, appending to a list is a sequential operation, which is unsuitable for applications that need multiple independent actors to grow the list simultaneously.
- The **key-ordered** linked list allows nodes to be inserted anywhere in the list, but it applies additional rules to maintain the key-ascending order of nodes. It can be used to implement sets and key-value maps. List insertion is increasingly parallel as the list grows, so the key-ordered linked list is well-suited to applications with multiple independent actors growing the list (see Section A.1.12).

A.1.1 Utxo representation

Each linked list, whether key-ordered or key-unordered, is represented in the blockchain ledger as a collection of node utxos that hold node NFTs minted by the list's minting policy and are held under the list's spending validator. The spending validator is parameterized by the minting policy, and the minting policy is parameterized by the utxo reference of one of the spent inputs in the transaction that initializes the list.

Each node utxo of a list uniquely corresponds to some `ByteArray` key and holds a node NFT with a token name equal to that key's serialization. The key-ordered list's state transitions guarantee that all nodes have unique keys, but the key-unordered list's state transitions only assume that keys are unique.

An application using a key-unordered list MUST only add nodes with unique keys to the list. Duplicate keys break the linked list data structure.

Every node utxo has a datum of the `NodeDatum` type, which is parametric on the `app_data` type of the non-key data that is to be stored in the list nodes:

$$\text{NodeDatum}(\text{app\_data}) := \left\{ \begin{array}{ll} \text{key}: & \text{Option}(\text{ByteArray}) \\ \text{link}: & \text{Option}(\text{ByteArray}) \\ \text{data}: & \text{app\_data} \end{array} \right\}$$

The `key` field indicates the node's key (or `None` for the root node). [1] It must match the token name of the node's NFT when serialized:[2]

$$\begin{aligned} \text{serialize\_key}(\text{None}) &:= \text{“Node”} \\ \text{serialize\_key}(\text{Some}(\text{key})) &:= \text{concat}(\text{“Node”}, \text{key}) \end{aligned}$$

The root node of a list is the only node with a `key` field value of `None` in its `NodeDatum` and its NFT token name set to `''Node''`. The root node is created when a list is initialized and must continue to exist until the list is deinitialized. This allows an initialized but empty list to be represented by just its root node (and no other nodes). The non-root nodes represent the actual keys and data of interest within a non-empty list.

The `link` field is a link to the next key that a forward traversal through the list must visit after visiting the current node. Alternatively, the `link` field can be interpreted as the previous key that a backward traversal through the list has visited before the current node. The last node is the only node with a `link` field value of `None` because forward traversal through the list must end after this node (backward traversal must start at this node).

The `data` field is parametric on the `app_data` type set by the application that instantiates the list. It is reserved for use by the application's custom onchain code.

A.1.2 Linked list library structure

The linked list library provides a collection of *rules* that enforce the baseline behavior of linked lists. An application using the list data structure should invoke one of these rules whenever it needs a particular linked list operation to occur within the context of the application's custom state transitions.

[1] The key field is redundant, as it is already represented in the node NFT, but we include it in the datum for convenience in onchain scripts and offchain analytics.

[2] Prefixing the key when it is serialized to the token name prevents a collision between the root node and a node with an empty key string. It also allows more flexibility for minting policies that use the linked list library to atomically mint additional tokens associated with the key but namespaced from the node NFTs. Alternatively, if this functionality is not needed, the "Node" prefixes can be removed from the serialization.

Linked list rules are predicate functions that can be freely composed with any custom application rules via logical conjunction. In this way, while the linked list rules control which operations can be performed on a list and how they must be performed, applications can apply further constraints on the circumstances under which the operations are allowed.

The state transition rules control how lists are initialized/deinitialized and how nodes are added/removed from the list. These rules should be invoked in the application's minting policy for the list because every state transition of the list changes the number of list nodes, so it must necessarily mint or burn node NFTs.

The state integrity rules ensure that node NFTs stay in their respective node utxos until burned and that the `key` and `link` fields remain unchanged outside of linked list state transitions. These rules should be invoked in the application's spending validator for the list.

Neither the state transition nor the state integrity rules inspect the `data` field of node utxos, nor do they inspect the value held by node utxos besides node NFTs. Applications are free to use these as they wish.

Section A.1.10 describes how to instantiate a linked list within an application and compose the linked list rules with custom app rules.

A.1.3 State integrity rules

The state integrity rules distinguish between transactions that apply linked list state transitions and transactions that merely modify the `data` field or non-node-NFT value of a node utxo.

List State Transition. This rule allows a node utxo to be spent in any transaction that applies a state transition to the list. Conditions:

1. Tokens of the list's minting policy must be minted or burned.

Modify Data. This rule ensures that the node NFT, the `key` field, and the `link` field remain unchanged in any transaction that does not apply a state transition to the list. It achieves it as follows:

1. Tokens of the list's minting policy must *not* be minted or burned.
2. Let `own_input` be the node utxo input for which this rule is being evaluated.
3. Let `own_output` be an output of the transaction indicated by a redeemer argument of the spending validator.
4. The node NFT of the list must be present in `own_input`.
5. The value must match in `own_input` and `own_output`.
6. The `key` and `link` fields must match in `own_input` and `own_output`.

To keep the onchain code simple and efficient, this rule can only handle content modifications to one node utxo per list. In principle, this could be generalized to handle bulk modifications of node contents, but that would require more expensive computations to match node inputs with node outputs.

A.1.4 List subcontext for state transition rules

State transition rules of a linked list are only concerned with a transaction's effects on the list's node utxos. Consequently, they focus on the following linked list subcontext, which is a subset of the overall transaction context:

- `node_cs` is the minting policy ID of the list.
- `node_mint` is the map of tokens of the list's minting policy that were minted or burned.
- `node_inputs` are the transaction inputs that hold node NFTs of the list and datums that parse as `NodeDatum`.
- `node_outputs` are the transaction outputs that hold node NFTs of the list and datums that parse as `NodeDatum`.

The state transition rules ignore the rest of the transaction context, which does not affect the state of the list. However, applications invoking the state transition rules are free to consider the whole transaction context to constrain the circumstances under which they allow linked list state transitions to occur.

A.1.5 Node and key predicates

The following predicate functions characterize nodes and keys in a linked list subcontext relative to the global state of the list.

Root node. The root node of the list:

$$\mathsf{is\_root\_node}(\mathsf{node}) \coloneqq (\mathsf{node.datum.key} \equiv \mathsf{None})$$

Last node. The last node of the list:

$$\mathsf{is\_last\_node}(\mathsf{node}) \coloneqq (\mathsf{node.datum.link} \equiv \mathsf{None})$$

Empty list. The list is empty if its root node is also its last node:

$$\mathsf{is\_root\_node}(\mathsf{node}) \land \mathsf{is\_last\_node}(\mathsf{node})$$

Key added. The transaction only affects the list by adding the given key:

$$\mathsf{key\_added}(\mathsf{key}, \mathsf{node\_cs}, \mathsf{node\_mint}) \coloneqq \Big(\mathsf{without\_lovelace}(\mathsf{node\_mint}) \equiv \mathsf{from\_asset}(\mathsf{node\_cs}, \mathsf{serialize\_key}(\mathsf{key}), 1)\Big)$$

Key removed. The transaction only affects the list by removing the given key:

$$\mathsf{key\_removed}(\mathsf{key}, \mathsf{node\_cs}, \mathsf{node\_mint}) \coloneqq \Big(\mathsf{without\_lovelace}(\mathsf{node\_mint}) \equiv \mathsf{from\_asset}(\mathsf{node\_cs}, \mathsf{serialize\_key}(\mathsf{key}), -1)\Big)$$

Key membership. The given key is a member of the list, proved by the existence of a witness node that satisfies the following predicate for the key.

$$\text{is\_member}(\text{key}, \text{node}) := (\text{key} \equiv \text{node.datum.key})$$

- When the above predicate is satisfied by a transaction input, it means that the key is a member of the list immediately before the transaction.
- When the above predicate is satisfied by a transaction output, it means that the key is a member of the list immediately after the transaction.

Key non-membership (ordered lists only!). The given key is *not* a member of the list, as proved by the existence of witness a node that satisfies the following predicate for the key:

$$\begin{aligned}
&\text{is_non_member}(\text{None}, _\text{node}) := \text{False} \\
&\text{is_non_member}(\text{Some}(\text{key}), \text{node}) := \\
&\quad \Big(\text{is_root_node}(\text{node}) \vee (\text{node.datum.key} < \text{Some}(\text{key}))\Big) \\
&\qquad \wedge \Big(\text{is_last_node}(\text{node}) \vee (\text{Some}(\text{key}) < \text{node.datum.link})\Big)
\end{aligned}$$

- When the above predicate is satisfied by a transaction input, it means that the key is not a member of the list immediately before the transaction.
- When the above predicate is satisfied by a transaction output, it means that the key is not a member of the list immediately after the transaction.

In human terms, the key non-membership proof describes four possible scenarios in which a key is not a member of a key-ordered list:

1. The list is empty.
2. The root node links to a node with a higher key than the given key, which precludes the given key's membership because all subsequent nodes visited after this node will have monotonically increasing keys.
3. The last node has a key lower than the given key, which precludes the given key's membership because all nodes visited before the last node have even smaller keys.
4. A non-root non-last node has a lower key but links to a higher key than the given key, which precludes the given key's membership because all nodes before this node have smaller keys, and all nodes after this node have larger keys.

For a key-unordered list, the only way to prove the non-membership of a key is to iterate over the entire list in the transaction inputs, which quickly becomes infeasible for lists larger than the empty list. The `is_non_member` predicate does *not* apply to key-unordered lists.

A.1.6 Indexing into a list

The main way to index into a list is by key. Every linked list, whether key-ordered or key-unordered (if its keys are unique), has a canonical surjective mapping from its domain of all possible keys onto the nodes it contains:

- If a given key is a member of the list, it is mapped to the node that proves its membership.
- Otherwise, the key is mapped to the node that proves its non-membership.

In the offchain context, indexing by key requires traversing the list to the node that proves the key's membership or non-membership. However, indexing by key is cheap in the onchain context because only that single node needs to be an input to the transaction.

The other way to index into a list is by position, which requires traversing the list to the node at that position in both the offchain and onchain contexts. The root and last nodes are the cheapest to reach because they intrinsically define their positions within the list, so inspecting any other nodes is unnecessary.

The first and second-last nodes are the next cheapest to reach because they are adjacent to the root and last nodes, respectively. This means that the root node must be an input when indexing into the first node, while the last node must be an input when indexing into the second-last node. The farther a node is from the root or last node, the larger this chain of inputs grows to establish its position in the list.

A.1.7 Onchain traversal of a list

In the onchain context, read-only traversal through a list is more efficient backwards than forwards when the only thing that the traversal needs to know about the previously visited node is its key. In that case, backward traversal only needs the current node to be a transaction input because the current node's `link` field matches the key of the previously visited node.

By contrast, read-only forward traversal requires both nodes to be transaction inputs because it only knows which node it should be visiting by inspecting the previously visited node's `link` field. Of course, the traversal state could instead keep a copy of the `link` field of the previously visited node, but that copy can become stale if the list is not kept immutable during the traversal.

On the other hand, forward traversal is more efficient when the list is destructively folded into an accumulator that is stored at the root node. In that case, the root node needs to be updated anyway, and it always points to the next node that the traversal should visit.

A.1.8 Key-unordered linked list

The key-unordered linked list data structure does *not* keep its nodes ordered by their keys. This means that a new node cannot be added to the list based on its key because the list does not require the key to be positioned after any particular node in the list.

This leaves only adding new nodes to the list at specific absolute positions. We restrict these positions even further to just the beginning and end of the list because other positions would require increasingly larger chains of reference inputs to establish the new node's position.

Therefore, the key-unordered linked list data structure allows new nodes to be added to the beginning or end of the list, while non-root nodes can be removed anywhere in the list. It enforces the following properties:

1. The root node exists continuously until the list is initialized.
2. If the keys in the list are unique, then each node has only one node linking to it.
3. If the keys in the list are unique, then traversal through the list is deterministic.

State transition rules

Init. Initialize an empty list. Conditions:

1. The transaction's sole effect on the list is to add the root key.

$$\text{key\_added}(\text{None}, \text{node\_cs}, \text{node\_mint})$$

2. The list must be empty after the transaction, as proved by an output `root_node` that holds the minted root node NFT.

$$\text{is\_empty\_list}(\text{root\_node}) \wedge \text{has\_token}(\text{root\_node}, \text{node\_cs}, \text{“Node”})$$

3. The `root_node` must not contain any other non-ADA tokens.

The above conditions imply the following:

- `node_outputs` must be a singleton. This is implied by the list being empty after the transaction, the root node NFT being minted, and no other node tokens being minted or burned.
- `node_inputs` is empty. This is implied by the root node NFT of the list being minted in the transaction. If list keys are unique, no other node can exist before the root node. This is enforced by the other state transition rules, which (inductively) require the existence of the root node before and after any other node NFTs are created or burned.

The Init rule cannot enforce that the root node utxo is sent to the spending validator that contains the state integrity rules of the list because that spending validator is parametrized on the minting policy that includes the Init rule.

Offchain code for the list-initialization transaction MUST send the root node to the list's spending validator. Otherwise, the linked list can be corrupted.

Deinit. Deinitialize an empty list. Conditions:

1. The transaction's sole effect on the list is to remove the root key.

$$\text{key\_removed}(\text{None}, \text{node\_cs}, \text{node\_mint})$$

2. The list must be empty before the transaction, as proved by an input `root_node` that holds the minted root node NFT.

$$\text{is\_empty\_list}(\text{root\_node}) \wedge \text{has\_token}(\text{root\_node}, \text{node\_cs}, \text{“Node”})$$

The above conditions imply the following:

- `node_inputs` must be a singleton. This is implied by the empty list before the transaction.
- `node_outputs` must be empty. This is implied by the condition that the root node NFT is burned and that no other node tokens are minted or burned.

Prepend (unsafe). Prepend a new node to the beginning of the list. Grouped conditions:

- Verify the mint:
  1. Let `key_to_prepend` be the key being prepended.
  2. The transaction's sole effect on the list is to add `key_to_prepend`.

$$\text{key\_added}(\text{key\_to\_prepend}, \text{node\_cs}, \text{node\_mint})$$

- Verify the inputs:
  3. `node_inputs` must be a singleton. Let `anchor_node_input` be its sole node.
  4. `anchor_node_input` must be the root node of the list.
- Verify the outputs:
  5. `node_outputs` must have exactly two nodes:
     - `prepended_node`
     - `anchor_node_output`
  6. `key_to_prepend` must be a member of the list after the transaction, as witnessed by `prepended_node`.

$$\text{is\_member}(\text{key\_to\_prepend}, \text{prepended\_node})$$

  7. `anchor_node_input` and `prepended_node` must match on the `link` field. In other words, they must both link to the same key.
  8. `anchor_node_output` must link to `key_to_prepend`.
  9. `prepended_node` must not contain any other non-ADA tokens.
- Verify immutable data:
  10. `anchor_node_input` must match `anchor_node_output` on address, value, and datum except for the `link` field.
  11. `prepended_node` must match `anchor_node_output` on address.

This rule is considered unsafe because it does *not* enforce key uniqueness or key order in the list. It merely assumes that the key being prepended is unique.

An application using a key-unordered list MUST only add nodes with unique keys to the list. Duplicate keys break the linked list data structure.

Append (unsafe). Append a new node to the end of the list. Conditions:

- Verify the mint:
  1. Let `key_to_append` be the key being appended.
  2. The transaction's sole effect on the list is to add `key_to_append`.

$$\text{key\_added}(\text{key\_to\_append}, \text{node\_cs}, \text{node\_mint})$$

- Verify the inputs:
  3. `node_inputs` must be a singleton. Let `anchor_node_input` be its sole node.
  4. `anchor_node_input` must be the last node of the list before the transaction.
- Verify the outputs:
  5. `node_outputs` must have exactly two nodes:
     - `appended_node`
     - `anchor_node_output`
  6. `key_to_append` must be a member of the list after the transaction, as witnessed by `appended_node`.

$$\text{is\_member}(\text{key\_to\_append}, \text{appended\_node})$$

  7. `appended_node` must be the last node of the list after the transaction.
  8. `anchor_node_output` must link to `key_to_append`.
  9. `appended_node` must not contain any other non-ADA tokens.
- Verify immutable data:
  10. `anchor_node_input` must match `anchor_node_output` on address, value, and datum except for the `link` field.
  11. `appended_node` must match `anchor_node_output` on address.

This rule is considered unsafe because it does *not* enforce key uniqueness or key order in the list. It merely assumes that the key being appended is unique.

An application using a key-unordered list MUST only add nodes with unique keys to the list. Duplicate keys break the linked list data structure.

Remove. Remove a non-root node from the list. Conditions:

- Verify the mint:
  1. Let `key_to_remove` be the key being removed.

2. The transaction's sole effect on the list is to remove `key_to_remove`.

$$\text{key\_removed}(\text{key\_to\_remove}, \text{node\_cs}, \text{node\_mint})$$

- Verify inputs:

3. `node_inputs` must have exactly two nodes:
   - `removed_node`
   - `anchor_node_input`
4. `key_to_remove` must be a member of the list before the transaction, as witnessed by `removed_node`.

$$\text{is\_member}(\text{key\_to\_remove}, \text{removed\_node})$$

5. `anchor_node_input` must link to `key_to_remove`.

- Verify outputs:

6. `node_outputs` must be a singleton. Let `anchor_node_output` be its sole node.
7. `anchor_node_output` and `removed_node` must match on the `link` field. In other words, they must both link to the same key.

- Verify immutable data:

8. `anchor_node_input` must match `anchor_node_output` on address, value, and datum except for the `link` field.

A.1.9 Key-ordered linked list

The key-ordered linked list data structure replaces the unsafe Append rule of the key-unordered linked list with the safe Insert rule. It re-uses the rest of the key-unordered linked list rules, as these rules cannot cause a key-ordered list to become key-unordered. It enforces the following properties:

1. The root node exists continuously until the list is deinitialized.
2. Each node has a unique key.
3. Each key has only one node linking to it.
4. Traversal through the list is deterministic and follows key-ascending order.

State transition rules

Init. Same as Init for key-unordered lists.

Offchain code for the list-initialization transaction MUST send the root node to the list's spending validator. Otherwise, the linked list can be corrupted.

Deinit. Same as Deinit for key-unordered lists.

Prepend (safe). Same as Prepend for key-unordered lists, but with an additional condition:

- `key_to_prepend` must *not* be a member of the list before the transaction, as witnessed by `anchor_node_input`.

$$\text{is\_not\_member}(\text{key\_to\_prepend}, \text{anchor\_node\_input})$$

The above condition enforces key uniqueness and order in the list, making this version of Prepend safe.

Append (safe). Same as Append for key-unordered lists, but with an additional condition:

- `key_to_append` must *not* be a member of the list before the transaction, as witnessed by `anchor_node_input`.

$$\text{is\_not\_member}(\text{key\_to\_append}, \text{anchor\_node\_input})$$

The above condition enforces key uniqueness and order in the list, making this version of Append safe.

Insert. Insert a node into the list. Grouped conditions:

- Verify mint:
  1. Let `key_to_insert` be the key being inserted.
  2. The transaction's sole effect on the list is to add `key_to_insert`.

$$\text{key\_added}(\text{key\_to\_insert}, \text{node\_cs}, \text{node\_mint})$$

- Verify inputs:
  3. `node_inputs` must be a singleton. Let `anchor_node_input` be its sole node.
  4. `key_to_insert` must *not* be a member of the list before the transaction, as witnessed by `anchor_node_input`.

$$\text{is\_not\_member}(\text{key\_to\_insert}, \text{anchor\_node\_input})$$

- Verify outputs:
  5. `node_outputs` must have exactly two nodes:

- `inserted_node`
- `anchor_node_output`

6. `key_to_insert` must be a member of the list after the transaction, as witnessed by `inserted_node`.

$$\text{is\_member}(\text{key\_to\_insert}, \text{inserted\_node})$$

7. `anchor_node_output` must link to `key_to_insert`.
8. `anchor_node_input` and `inserted_node` must match on the `link` field. In other words, they must both link to the same key.
9. `inserted_node` must not contain any other non-ADA tokens.

- Verify immutable data:

10. `anchor_node_input` must match `anchor_node_output` on address, value, and datum except for the `link` field.
11. `inserted_node` must match `anchor_node_output` on address.

Unlike the Append rule, the Insert rule is safe because it enforces key uniqueness and key order in the list:

- Key uniqueness is achieved by the inserted key's pre-transaction non-membership.
- Key order is achieved by requiring both `anchor_node_input` and `inserted_node` to link to the same key. The key non-membership condition implies that this common linked key is higher than `key_to_insert`, which means that `inserted_node` complies with the key order.

Remove. Same as Remove for key-unordered lists.

A.1.10 Instantiate a linked list in an application

An application that uses a linked list should ensure that it uses the appropriate state transition rules when creating/destroying node utxos of the list or otherwise modifying the `key` or `link` field value of any nodes. The application can attach additional rules that constrain the circumstances under which it allows list state transitions based on the list subcontext or the full transaction context. Moreover, the application has complete and exclusive control over the app-specific `data` field of list nodes, for which it also defines the data type.

As a general good practice, applications should limit the number of different non-node tokens and the size of app-specific data placed into node utxo when new nodes are inserted and when app-specific data is modified. This avoids token dust and large data attacks that could result in deadlock when certain list operations cannot fit within transactions' compute, memory, or space constraints.

Midgard uses linked lists throughout its onchain architecture:

- Registered operators (Section 3.2.2)
- Active operators (Section 3.2.3)
- Retired operators (Section 3.2.4)
- State queue (Section 3.4)

A.1.11 Different data in root node

An application that needs to store different data in the root node than in non-root nodes of a list can use an app-specific type with multiple constructors (i.e., a sum type) in the `data` field of the list's `NodeDatum`.

```
MyNodeDatum := NodeDatum(MyAppData)
MyAppData := Root(MyRootData) | Node(MyNodeData)
```

The application's onchain code should ensure that root and non-root nodes always use their respective constructors.

A.1.12 Parallel insertions in key-ordered lists

For keys randomly sampled from an approximately uniform distribution (e.g., cryptographic public keys for wallets), insertion/removal from a key-ordered linked list is parallel to a degree proportional to the list's size, with parallelism increasing as the list grows.

An application may expect highly parallel traffic (e.g., from simultaneous interactions of independent users) before its list grows from its initially small size. This can be alleviated by using "separator" nodes in the list, which boost the list's parallelism by virtually occupying specific keys. Ideally, separators should be evenly spaced throughout the key space of the list.

```
MyNodeDatum := NodeDatum(MyAppDataWithSeps)
MyAppDataWithSeps := Root(MyRootData) | Node(MyNodeData) | Separator
```

Suppose the application needs to insert a node at a key occupied by a separator node. In that case, it will instead modify the node contents (see Section A.1.3) of the separator node to the intended `data` field value of the inserted node. Otherwise, node insertion works as usual.

The application can insert or remove separators as needed during the lifecycle of the list to regulate its parallelism.

A.2 Single-threaded state machine

A.2.1 Utxo representation

Each single-threaded state machine's current state is represented in the blockchain ledger as a single utxo:

- The spending validator of the utxo defines the possible transitions out of the current state.
- The utxo value contains a thread token corresponding to the state machine.
- The datum contains the output that the machine emitted upon entering the current state.

Each state transition of the state machine is executed via a separate blockchain transaction:

- The state machine's thread token must be unique within the transaction context.[3]
- The before-state is represented by the transaction input that contains the thread token.
- The after-state is represented by the transaction output that contains the thread token.
- The state transition's input is represented by the redeemer provided to the before-state's spending validator. If the spending validator defines several state transitions, the redeemer selects one for the transaction.

A.2.2 Minting policy

The thread token's minting policy defines the state machine's initial states, initialization procedure, final states, and termination procedure. It also implicitly defines the state machine's subgraph of reachable states, as each initial state's spending validator defines the transitions out of that state and, inductively, all further transitions out of the resulting states. Redeemers:

Initialize. Mint the thread token and send it to the spending validator address of one of the state machine's initial states. This redeemer receives an initial input that selects the initial state and provides additional information that can be referenced in subsequent state transitions.

The thread token's name should indicate the selected initial state and include a hash of the reference information provided in the initial input. If the state machine is deterministic, its thread token name identifies its unique path from initialization to the current state.

Finalize. Terminate the state machine normally if it is in one of the final states. Burn the thread token and (if needed) perform cleanup actions that are universally needed when terminating normally from any final state.

Cancel. Terminate the state machine exceptionally from any state. Burn the thread token and (if needed) perform cleanup actions that are universally needed when terminating exceptionally from any state.

[3] This avoids double-satisfaction issues during onchain validation of state transitions, as any transition's before and after states can be uniquely identified in any transaction. However, the state machine's thread token does *not* need to be globally unique across the blockchain ledger — it may be desirable to run several machines simultaneously, evolving their states in independent transaction chains.

A.2.3 Spending validators

If the state machine is non-deterministic, some of its spending validators define multiple possible transitions out of some states. The redeemers provided to these spending validators select the state transitions out of those states. Furthermore, the redeemers may provide additional arguments so that the spending validators have the context to decide whether to allow the selected state transitions.

Each spending validator is custom-written to express the specific logic of the possible transitions out of its state. For each of these transitions, the spending validator must include a condition that verifies the transformation of the input state into the corresponding output state:

$$\begin{aligned} \mathsf{verify\_transition}_{ij} &: (\mathsf{Input}_i, \mathsf{Output}_j, ..\mathsf{Args}_j) \to \mathsf{Bool} \\ \mathsf{verify\_transition}_{ij}(\mathsf{i}, \mathsf{o}, ..\mathsf{args}) &:= \Big(\mathsf{transition}_{ij}(\mathsf{i}, ..\mathsf{args}) \equiv \mathsf{o}\Big) \\ \mathsf{transition}_{ij} &: (\mathsf{Input}_i, ..\mathsf{Args}_j) \to \mathsf{Output}_j \end{aligned}$$

A.2.4 Compilation

Each spending validator can be either statically or dynamically parametrized on the spending validators into which its outbound state transitions lead.

- Static parametrization is preferred for state transitions that are expected to occur more frequently in typical executions of the state machine. A fancy way of expressing this is that state transitions should be statically parametrized along the maximally weighted acyclic subgraph of the state graph.
- Dynamic parametrization should be used for all other state transitions because statically parametrizing them would cause circular compilation dependencies on more preferred state transitions.

Dynamic parametrization means that the spending validator requires a reference input that indicates the addresses of the spending validators on which it dynamically depends. This reference input is crucial for the integrity of the state machine's state graph, so secure governance mechanisms should control its creation/modification.

A.2.5 Example

Consider a simplified model of the git pull-request (PR) workflow, with the following states:

Draft (initial state). The drafter is implementing a feature or bug fix in a repository branch. Transitions:

Update. The drafter updates the branch by adding some git commits. Next state: Draft.

Request review. The drafter requests a review for the branch. Next state: Testing.

Testing. The test suite is executing. Transitions:

Fail. The branch fails its test suite. Next state: Draft.

Pass. The branch passes its test suite. Next state: Review.

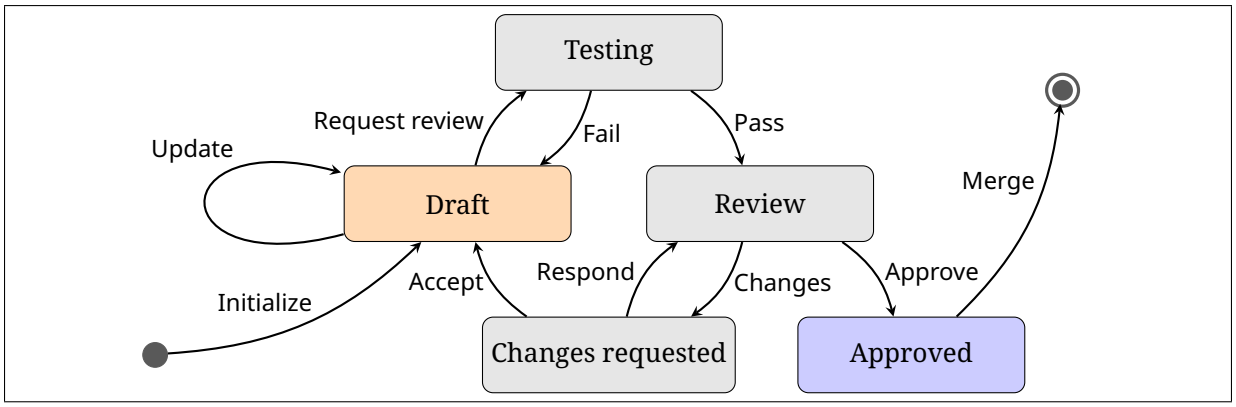

Figure A.2.1: State diagram for a simplified git PR workflow.

Review. The reviewer is deciding whether the branch should merge into the main branch. Transitions:

Approve. The reviewer approves the branch to be merged into the repository's main branch. Next state: Merged.

Changes. The reviewer requests some changes to the branch. Next state: Changes requested.

Changes requested. The drafter is considering the reviewer's feedback. Transitions:

Respond. The drafter responds to the reviewer, arguing that changes are not required. Next state: Review.

Accept. The drafter accepts the reviewer's change requests. Next state: Draft.

Approved (final state). The branch is merged into the main branch. Transitions:

Merge. The state machine terminates normally from the final state. The branch is merged into the main branch and then deleted.

The onchain state machine representation of the above git PR model uses one minting policy and five spending validators. The minting policy:

- Defines Draft as the initial state.
- Assigns the state machine a token name corresponding to the commit hash of the base branch of the PR.[4]
- Defines Merged as the final state.
- Updates the state of the main branch when the PR branch is merged.

The spending validators define the transitions out of their corresponding states. The state datum types include the PR's current commit hash and other information relevant to their outbound transitions. For example, the spending validator for the Draft state has redeemers to validate two state transitions:

Update. Go to the Draft state. Update the PR commit hash and resolve any accepted change requests that the update addresses.

[4] In this simplified model, the base branch cannot be changed for a PR.

Request review. Go to the Testing state. Ensure that no accepted change requests remain.

We can also add a Cancel redeemer to every spending validator and the minting policy. In the git PR workflow, this state transition would reflect the fact that a PR can be closed at any time. Some additional logic may be needed in each of these redeemers to properly dispose of the PR after it is closed.

Appendix B

Midgard L1 transactions

This chapter specifies the L1 transactions that interact with Midgard's consensus protocol smart contracts. Whereas the onchain smart contract are specified in a modular way, where each validator is only concerned with its own validity conditions, the transactions specified in this chapter often interact with several of the onchain validators and must satisfy all of their conditions. Thus, the offchain code can be seen as the integration layer for the onchain code.

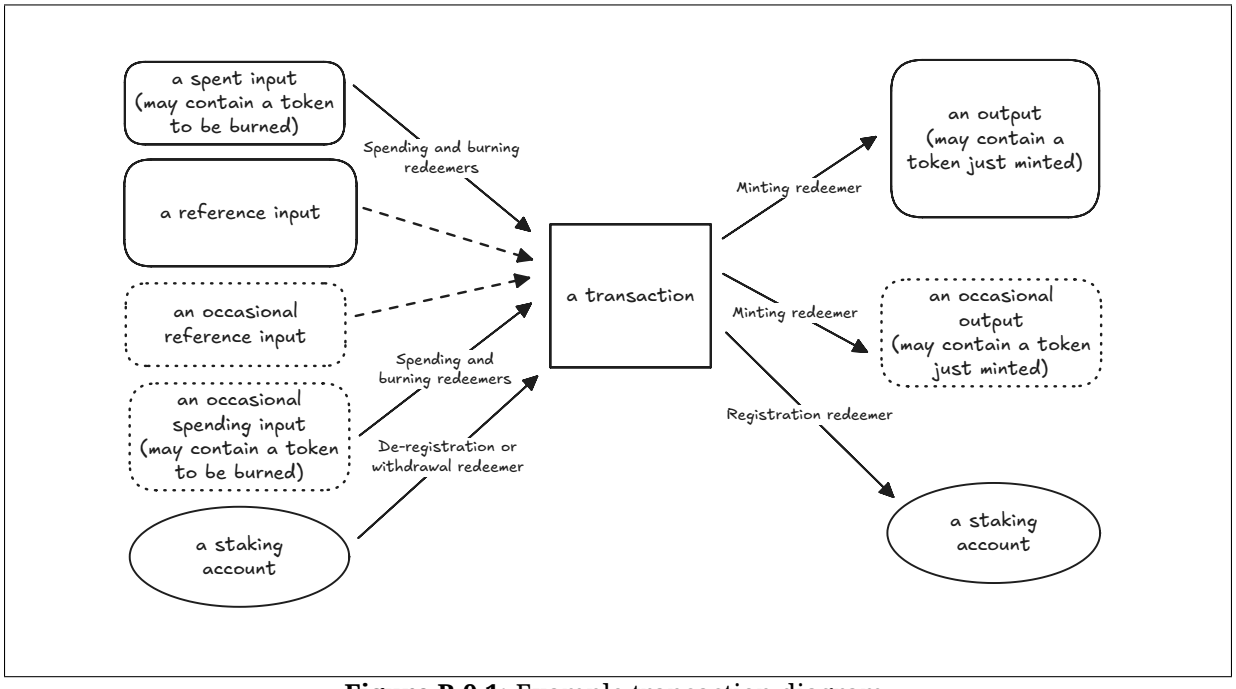

Figure B.0.1: Example transaction diagram.

As illustrated in the example diagram (Figure B.0.1), interpret our transaction diagrams as follows:

- Transactions are sharp-corner rectangles.
- Spent inputs are rounded-corner rectangles with outgoing arrows to transactions.
- Reference inputs are rounded-corner rectangles with dashed outgoing arrows to transactions.
- Outputs are rounded-corner rectangles with incoming arrows from transactions.

Branch: (None) @ 8799ca9 • Commit date: 2025-10-22

- Staking account are ellipses with incoming arrows from transactions if they're being registered and outgoing arrows to transactions if they're being de-registered or withdrawn from.
- Occasional inputs/outputs are rounded-corner rectangles with dotted borders. This indicates that these inputs/outputs may sometimes be present in transactions, but may be absent in other times.
- Spending redeemers label arrows from inputs to transactions.
- Burning redeemers label arrows from inputs to transactions, when those inputs contain tokens to be burned by the transactions.
- Minting redeemers label arrows from transactions to outputs, when those outputs contain tokens minted by the transactions.
- Redeemer labels may be omitted when scripts have only one possible redeemer.

B.1 Initialization

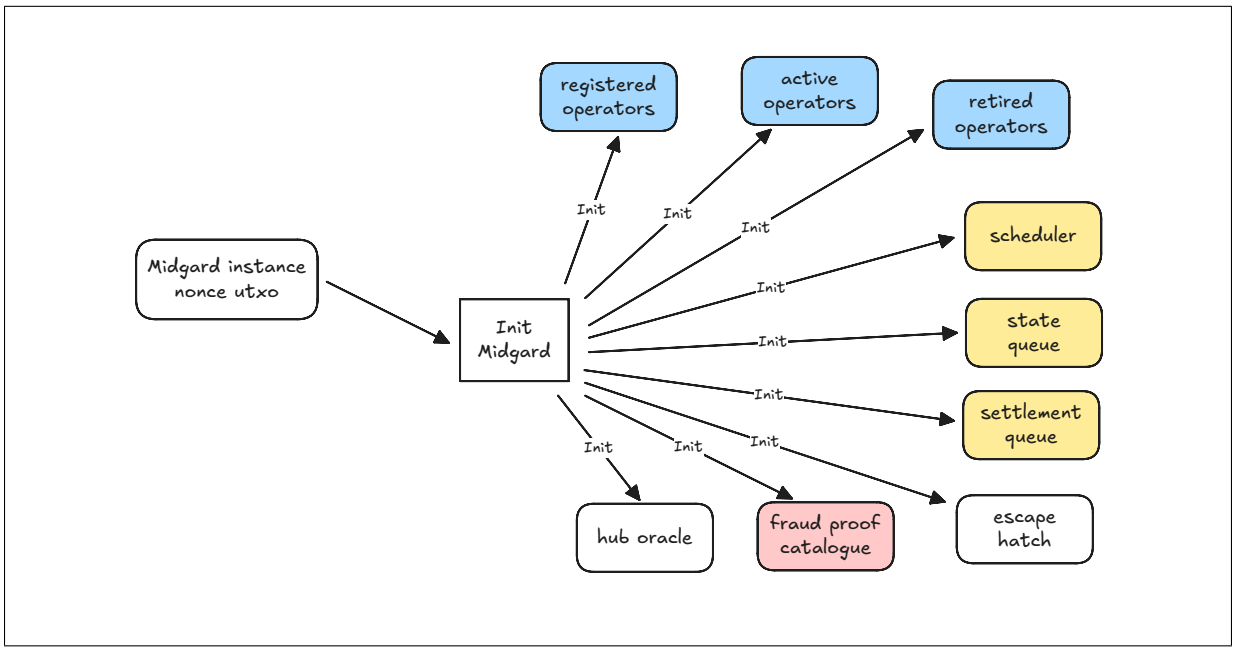

Figure B.1.1: Initialize Midgard.

Initialize a Midgard instance with a single transaction that sets up all the onchain data structures required for Midgard's smart-contract-based consensus protocol. This transaction (Figure B.1.1):

1. Spends a nonce utxo to uniquely parametrize[1] and authorize the initialization[2] of the Midgard instance.
2. Produces the root nodes of the initially empty registered, active, and retired operator lists.
3. Produces the root nodes of the initially empty state and settlement queues.

[1] A Midgard instance's hub oracle minting policy is parametrized on the nonce utxo spent in the initialization transaction, and all minting policies of the Midgard instance are parametrized on the hub oracle minting policy.

[2] For example, if the nonce utxo is spent from a native script address, then initializing a Midgard instance parametrized by that nonce utxo must satisfy the signature and timing requirements of the native script.

4. Produces the scheduler, escape hatch, fraud proof catalogue, and hub oracle singleton utxos.
5. Attaches the signatures necessary to spend the nonce utxo.

Correctness is enforced by the hub oracle minting policy's Init redeemer (Section 3.8.1).

B.2 Operator management

Midgard has several transactions that manage operators by interacting with the data structures of the operator directory.

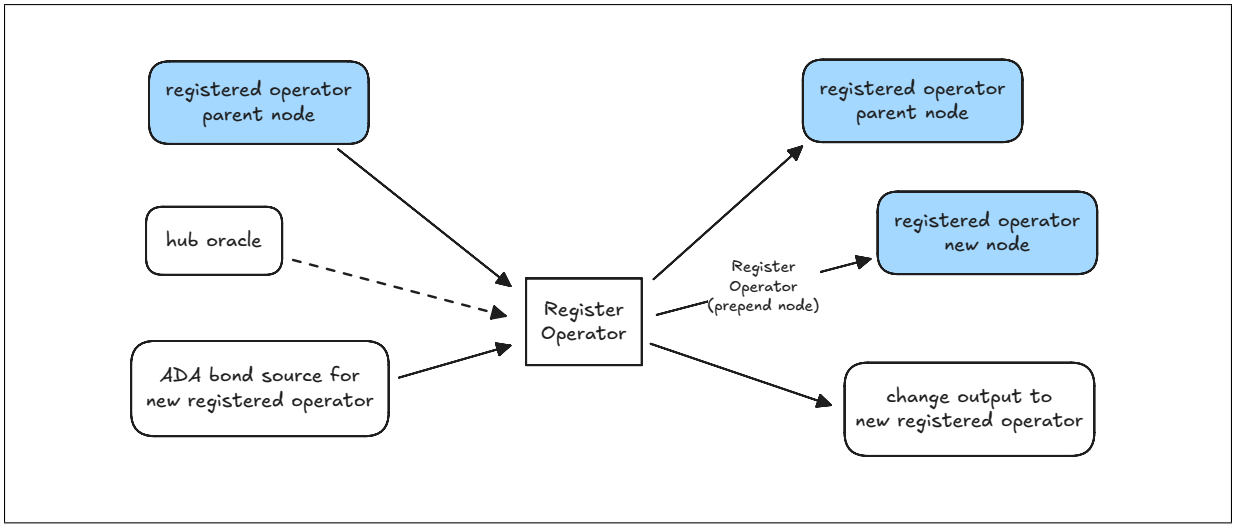

Figure B.2.1: Register an operator.

Register a new operator via a transaction that (Figure B.2.1):

1. Prepends a new operator node to the registered operators list.
2. Places a sufficient ADA bond into the new operator's node.
3. Attaches the operator's signature to the transaction.

Correctness is enforced by the registered operators minting policy's Register Operator redeemer (Section 3.2.2).

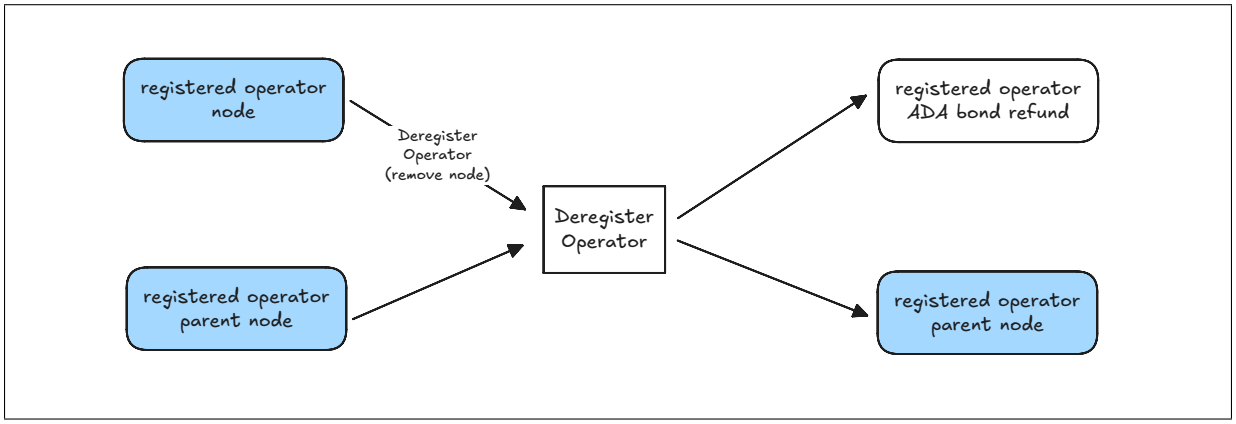

Figure B.2.2: De-register an operator.

De-register an operator via a transaction that (Figure B.2.2):

1. Removes the operator's node from the registered operator's list.
2. Refunds the ADA bond to the operator.
3. Attaches the operator's signature to the transaction.

Correctness is enforced by the registered operators minting policy's Deregister Operator redeemer (Section 3.2.2).

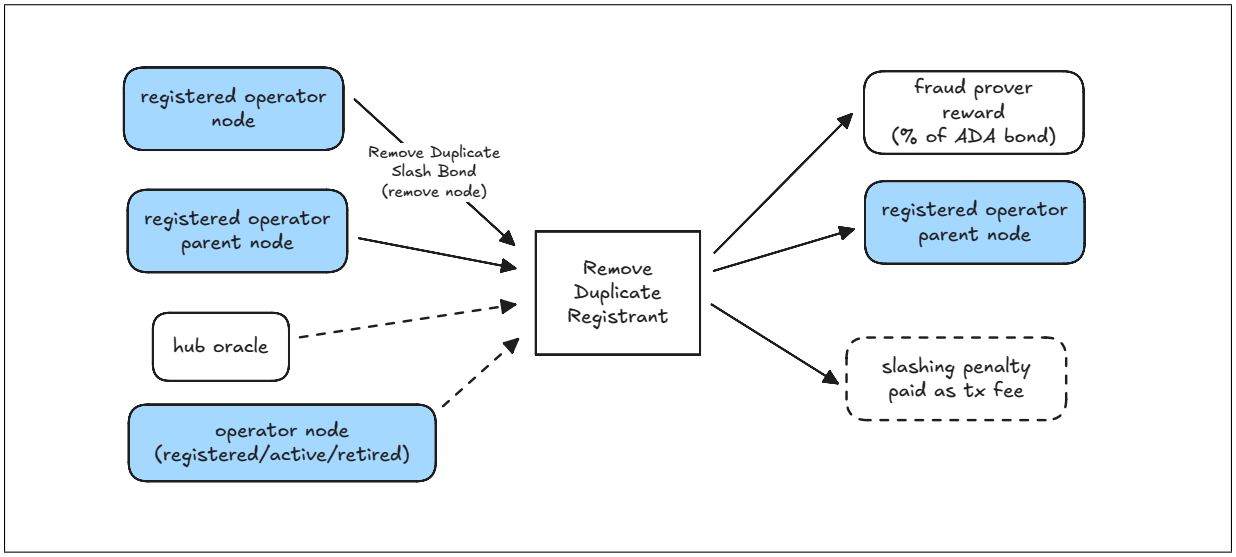

Figure B.2.3: Remove and slash a duplicate registered operator.

If the same operator key is simultaneously present in a registered operators node and in another node of the registered, active, or retired operators lists, then it can be removed via a transaction that (Figure B.2.3):

1. Removes the duplicate node from the registered operators list
2. Slashes the duplicate node's operator ADA bond, allocating part of it (`slashing_penalty` param) to the transaction's fee and allowing the transaction submitter to claim the rest as the fraud prover reward.
3. References the same operator key being used in another list node (registered/active/retired) to justify removing and slashing that operator's duplicate node.

Correctness is enforced by the registered operators minting policy's Remove Duplicate Slash Bond redeemer (Section 3.2.2).

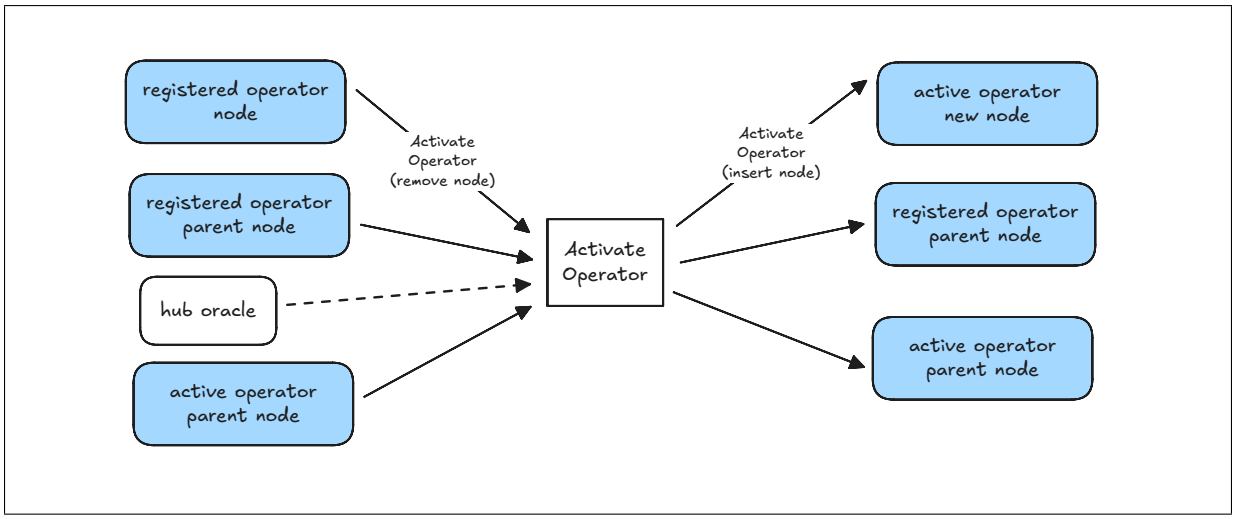

Figure B.2.4: Activate a registered operator.

Activate an operator via a transaction that (Figure B.2.4):

1. Removes the operator's node from the registered operators list.
2. Inserts a node for the operator in the active operators list.
3. Transfers the operator's ADA bond from the removed registered operator node to the inserted active operators node.

Correctness is jointly enforced by these script redeemers:

- Activate Operator of the registered operators minting policy (Section 3.2.2).
- Activate Operator of the active operators minting policy (Section 3.2.3).

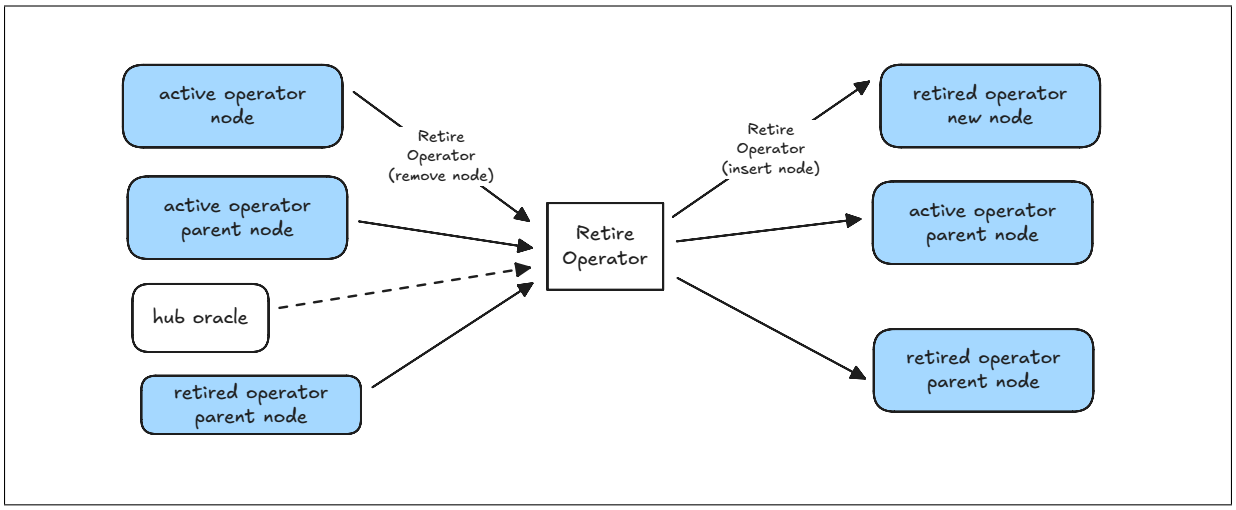

Figure B.2.5: Retire an active operator.

Retire an operator via a transaction that (Figure B.2.5):

1. Removes the operator's node from the active operators list.
2. Inserts a node for the operator in the retired operators list.

3. Transfers the operator's ADA bond from the removed active operator node to the inserted retired operator node.

4. Attaches the operator's signature to the transaction.

Correctness is jointly enforced by these script redeemers:

- Retire Operator of the active operators minting policy (Section 3.2.3).
- Retire Operator of the retired operators minting policy (Section 3.2.4).

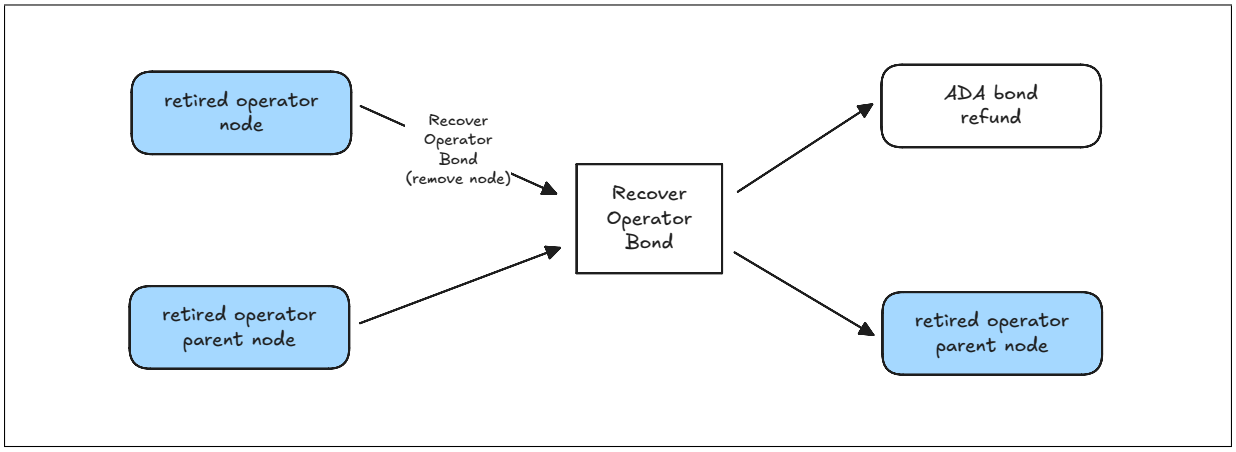

Figure B.2.6: Recover a retired operator's bond.

After all of a retired operator's block headers (in the state queue) and resolution claims (in the settlement queue) have matured, the operator can recover his ADA bond via a transaction that (Figure B.2.6):

1. Removes the operator's node from the retired operators list.
2. Refunds the operator's ADA bond to the operator.
3. Attaches the operator's signature to the transaction.

Correctness is enforced by the retired operators minting policy's Recover Operator Bond redeemer (Section 3.2.4).

B.3 Scheduler

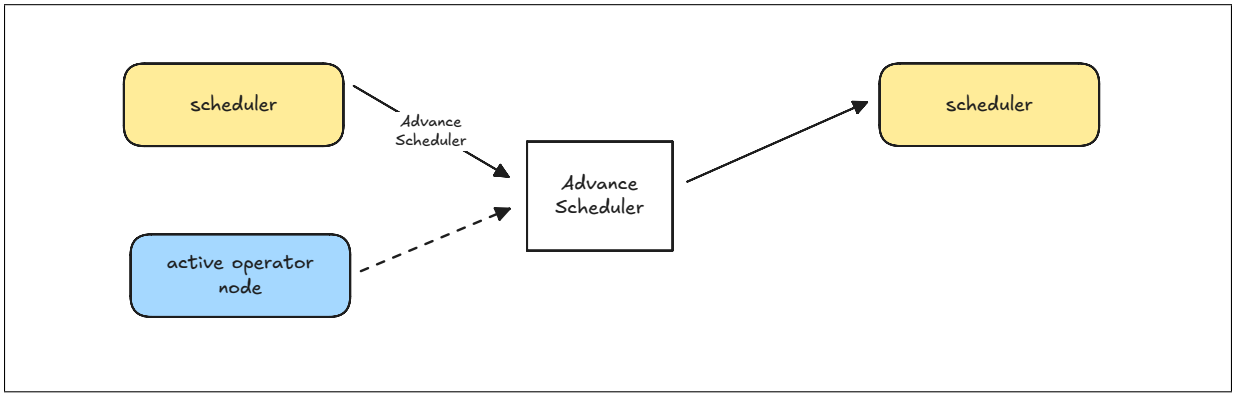

Figure B.3.1: Advance the scheduler to the next operator.

Advance the scheduler via transaction that (Figure B.3.1):

1. Updates the scheduler's datum to start the next shift and assign it to a new operator.
2. References an active operator node that proves that the newly assigned operator follows the previously assigned operator in key-descending order of the active operators list.

Correctness is enforced by the scheduler spending validator's Advance redeemer (Section 3.3.3).

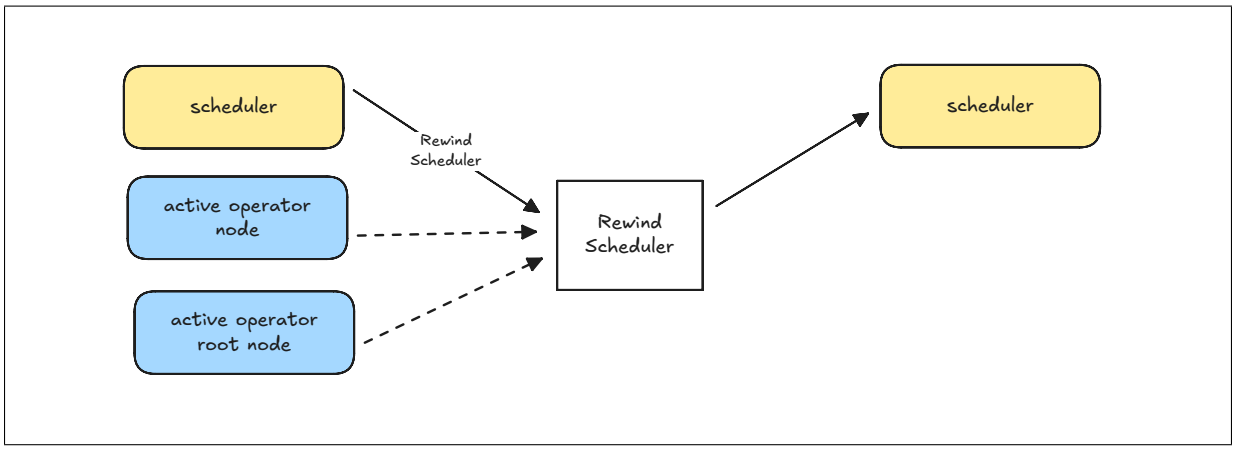

Figure B.3.2: Rewind the scheduler to the first operator.

Rewind the scheduler via a transaction that (Figure B.3.2):

1. Updates the scheduler's datum to the start the next shift and assign it to a new operator.
2. References the active operators' root and last nodes to prove that the previously assigned operator was at the end and the newly assigned operator is at the beginning of the key-descending order of the active operators list.

Correctness is enforced by the scheduler spending validator's Rewind redeemer (Section 3.3.3).

B.4 State queue

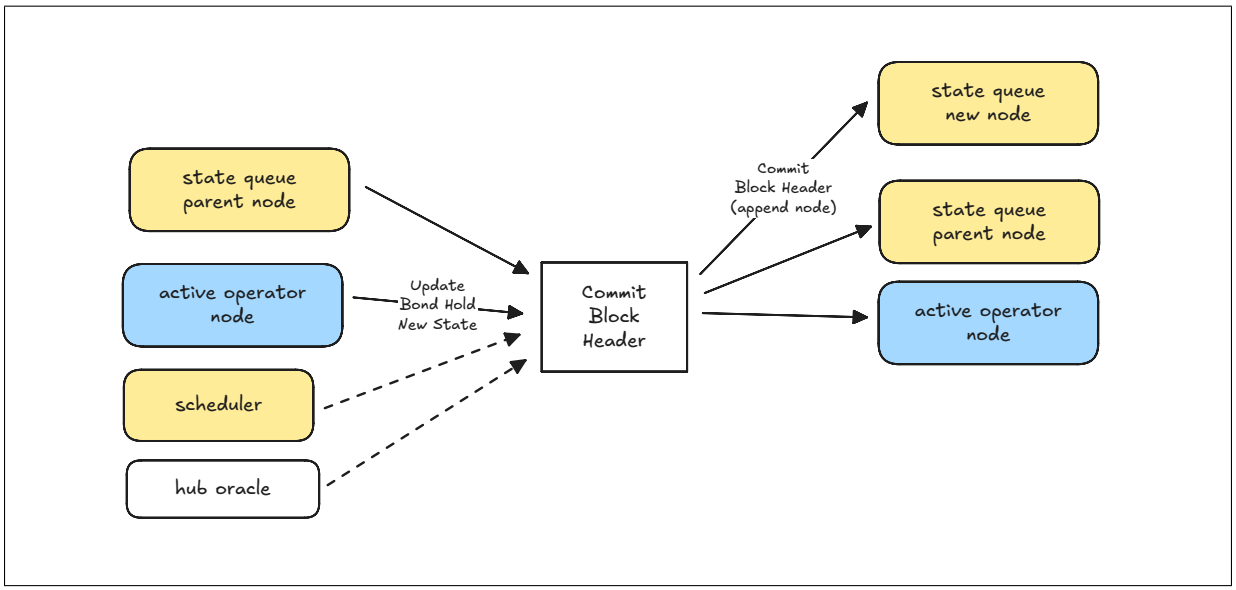

Figure B.4.1: Commit a block header to the state queue.

Commit a new block header to the state queue via a transaction that (Figure B.4.1):

1. Appends a new node containing the block header to the state queue.
2. Updates the operator's bond-unlock time to the future time when the new block header will mature.
3. Attaches the operator's signature to the transaction.
4. References the scheduler to verify that the operator has the right to commit a block during the current shift.

Correctness is jointly enforced by these script redeemers:

- Commit Block Header of the state queue minting policy (Section 3.4.2).
- Update Bond Hold New State of the active operators spending validator (Section 3.2.3).

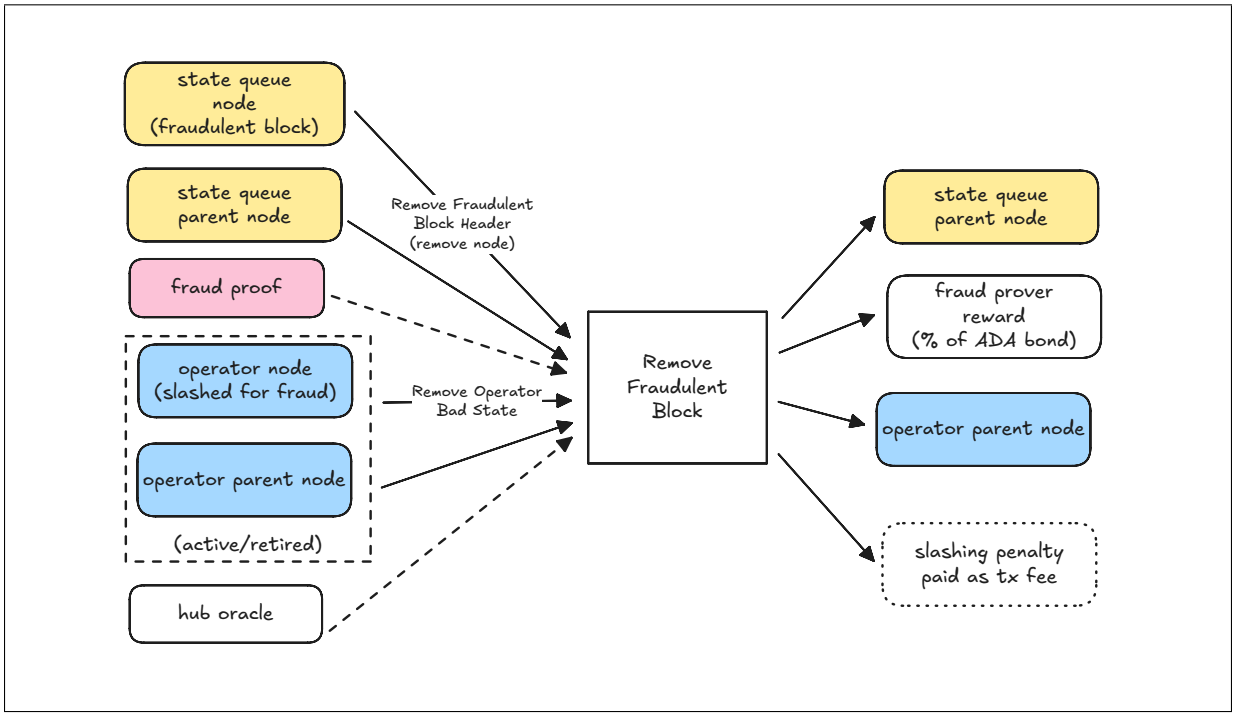

Figure B.4.2: Remove a fraudulent block header and slash its operator.

Remove a fraudulent block header from the state queue via a transaction that (Figure B.4.2):

1. References a fraud proof to justify removing the block header and slashing its operator.
2. Removes the fraudulent block header's node from the state queue.
3. Removes the dishonest operator's node from the active operators or retired operators list (depending on where the operator's node resides).
4. Slashes the dishonest operator's ADA bond, sending part of it (`fraud_prover_reward` param) to the fraud prover and paying the rest (`slashing_penalty`) as transaction fees.

Correctness is jointly enforced by these script redeemers:

- Remove Fraudulent Block Header of the state queue minting policy (Section 3.4.2).

- Remove Operator Bad State of the active/retired operators minting policy (Section 3.2.3, Section 3.2.4).

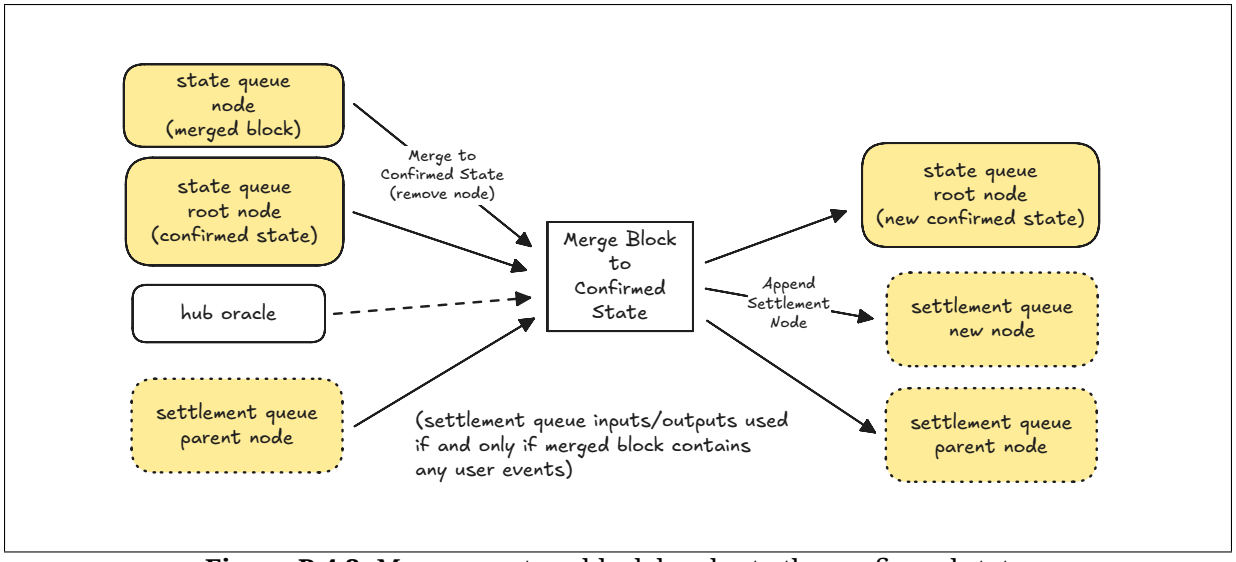

Figure B.4.3: Merge a mature block header to the confirmed state.

Merge a matured block header to the confirmed state via a transaction that (Figure B.4.3):

1. Removes the block header's node from the state queue.
2. Updates the state queue's root node datum (i.e. Midgard's confirmed state) to the reflect the information contained in the block header.
3. If the block header contains any L1 user events (deposits, transaction orders, or withdrawal orders) append a new node containing the same user events to the settlement queue.

Correctness is jointly enforced by these script redeemers:

- Merge to Confirmed State of the state queue minting policy (Section 3.4.2).
- Append Settlement Node of the settlement queue minting policy (Section 3.6.2).

B.5 Settlement queue

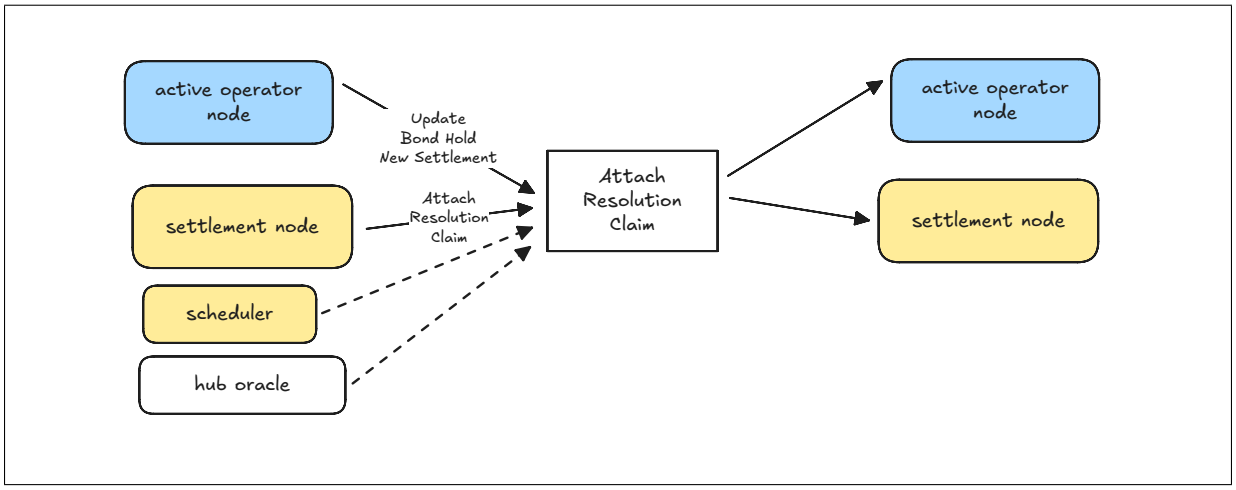

Figure B.5.1: Attach an optimistic resolution claim to a settlement node.

Attach a claim that all user events in a settlement node have been resolved, via a transaction that (Figure B.5.1):

1. Spends a settlement node without a resolution claim and updates its datum to attach a resolution claim. The claim must indicate the operator attaching the claim and the future time when the claim will mature (must be set according to `maturity_duration` param).
2. Spends the operator's node (must be active) and updates its datum's bond unlock time to the resolution claim's maturity time.
3. Attaches the operator's signature to the transaction.
4. References the scheduler to demonstrate that the current shift is assigned to the operator attaching the resolution claim.

Correctness is jointly enforced by these script redeemers:

- Attach Resolution Claim of the settlement queue spending validator (Section 3.6.3).
- Update Bond Hold New Settlement of the active operators spending validator (Section 3.2.3).

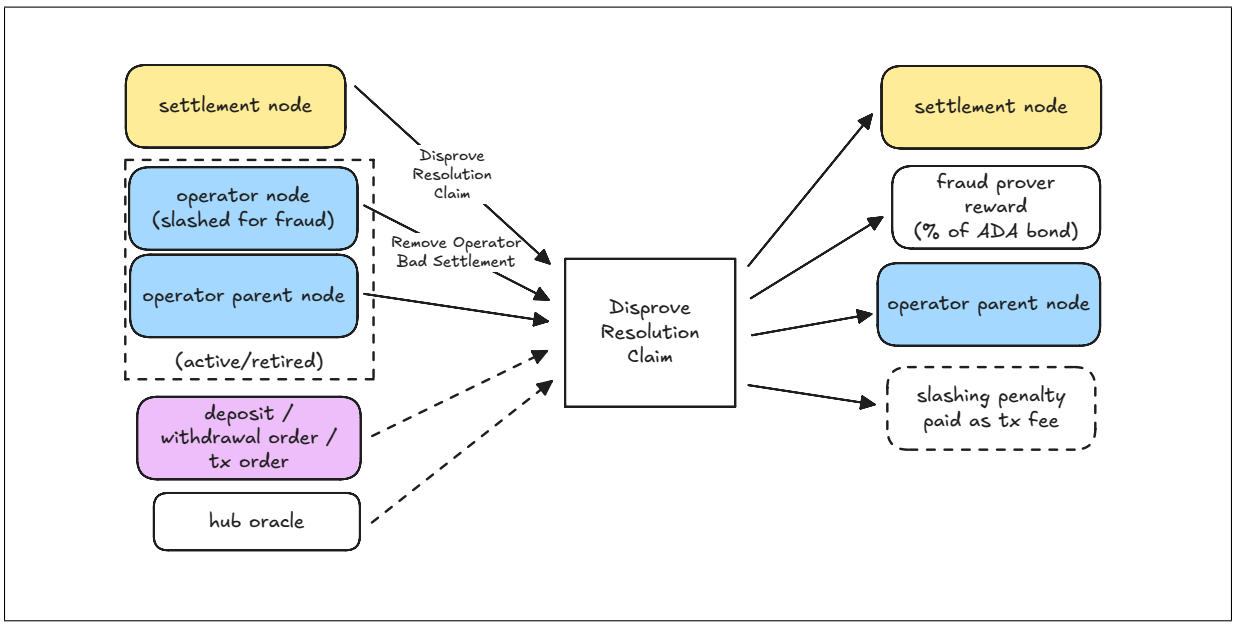

Figure B.5.2: Disprove an optimistic resolution claim of a settlement node.

Disprove a settlement node's resolution claim via a transaction that (Figure B.5.2):

1. Spends a settlement node with a resolution claim and updates it to remove the resolution claim.
2. Removes the resolution claim's operator node from the active operators or retired operators list (depending on where the operator's node resides).
3. Slashes the dishonest operator's ADA bond, sending part of it (`fraud_prover_reward` param) to the transaction's submitter and paying the rest (`slashing_penalty`) as transaction fees.
4. References an L1 user event as a counter-example to the resolution claim—the event is included in the settlement node but has not yet been resolved.

Correctness is jointly enforced by these script redeemers:

- Disprove Resolution Claim of the settlement queue spending validator (Section 3.6.3).
- Remove Operator Bad Settlement of the active operators or retired operators minting policy (Section 3.2.3, Section 3.2.4).

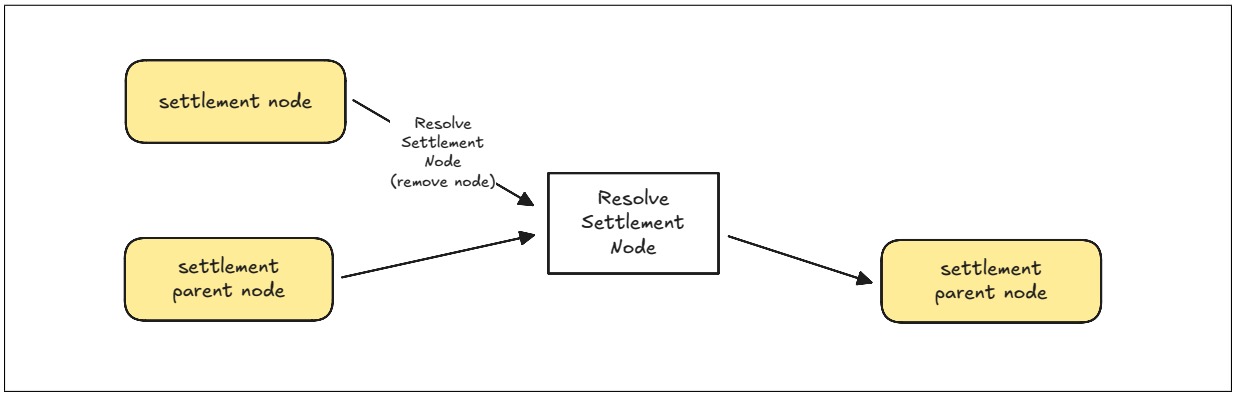

Figure B.5.3: Resolve a settlement node after its resolution claim matures.

Resolve a settlement node via a transaction that (Figure B.5.3):

1. Removes a settlement queue node that contains a resolution claim with a maturity time preceding the transaction's time-validity lower bound.
2. Attaches the signature of the resolution claim's operator.

Correctness is enforced by the Resolve Settlement Node script redeemer of the settlement queue's minting policy (Section 3.6.2).

B.6 User events

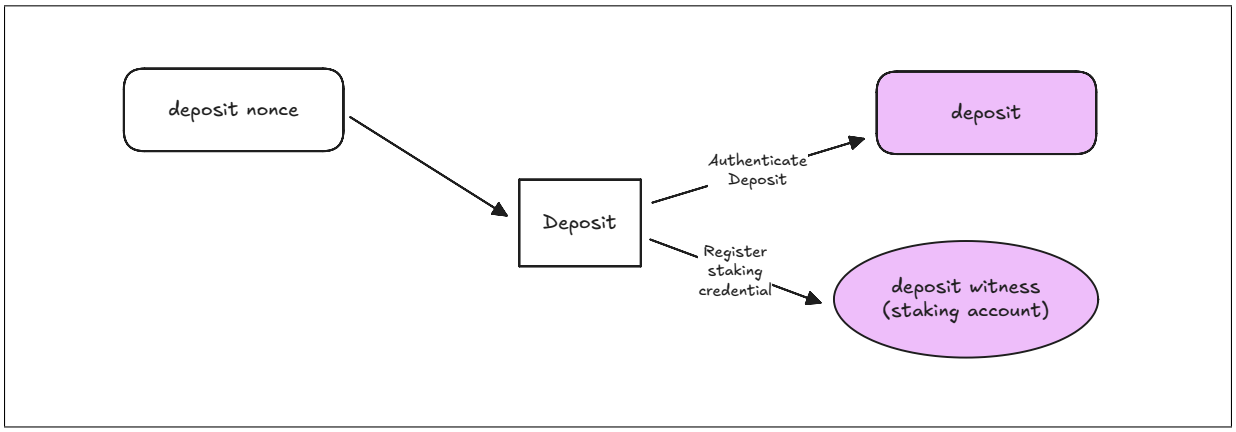

Figure B.6.1: Create an L1 deposit for Midgard.

Create a new deposit for Midgard via a transaction that (Figure B.6.1):

1. Spends a nonce input to uniquely parametrize the deposit.
2. Produces a deposit utxo containing the user's deposited funds and instructions.
3. Registers a staking credential to witness the existence of the deposit.

Correctness is jointly enforced by these script redeemers:

- Authenticate Deposit of the deposit minting policy (Section 2.1.1).
- Register Staking Credential of the witness staking script (Section 2.5.1).

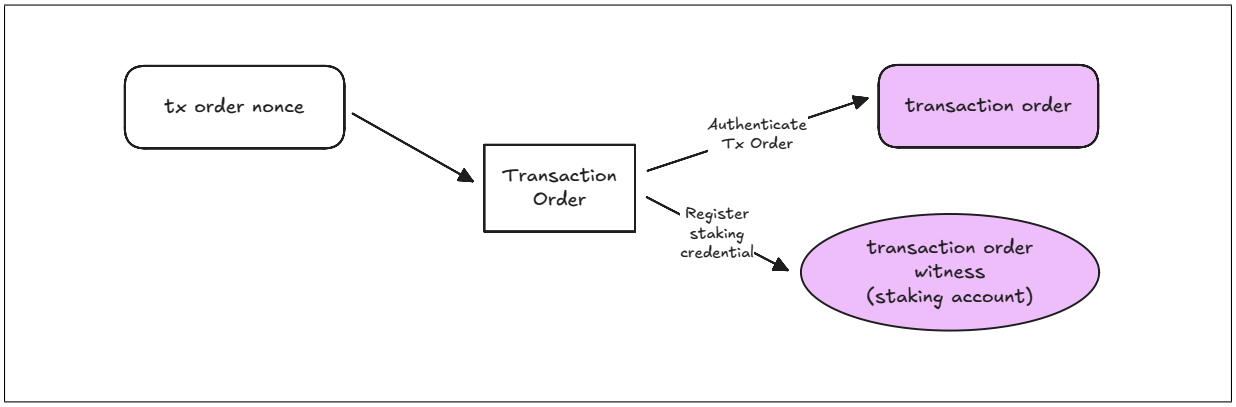

Figure B.6.2: Create a transaction order for Midgard.

Create a new transaction order for Midgard via a transaction that (Figure B.6.2):

1. Spends a nonce input to uniquely parametrize the transaction order.
2. Produces a transaction order utxo containing the user's L2 transaction.
3. Registers a staking credential to witness the existence of the transaction order.

Correctness is jointly enforced by these script redeemers:

- Authenticate Transaction Order of the transaction order minting policy (Section 2.3.1).
- Register Staking Credential of the witness staking script (Section 2.5.1).

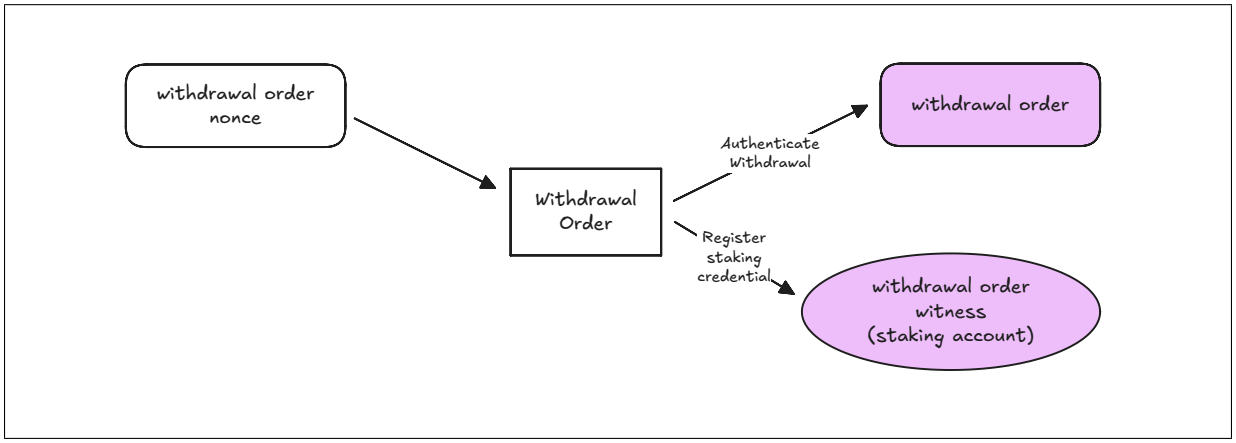

Figure B.6.3: Create a withdrawal order for Midgard.

Create a new withdrawal order for Midgard via a transaction that (Figure B.6.2):

1. Spends a nonce input to uniquely parametrize the withdrawal order.
2. Produces a withdrawal order utxo indicating the L2 utxo to be withdrawn.
3. Registers a staking credential to witness the existence of the withdrawal order.

Correctness is jointly enforced by these script redeemers:

- Authenticate Withdrawal Order of the withdrawal order minting policy (Section 2.3.1).
- Register Staking Credential of the witness staking script (Section 2.5.1).

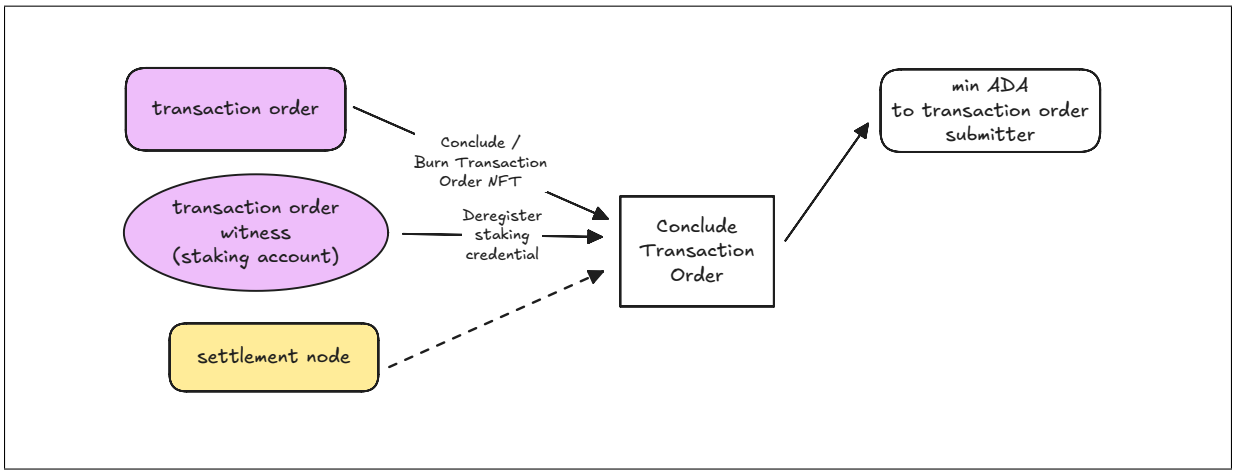

Figure B.6.4: Conclude a transaction order after it has been confirmed.

Conclude a confirmed transaction order via a transaction that (Figure B.6.4):

1. Spends a transaction order and burns its NFT.
2. De-registers the staking credential witnessing the transaction order.
3. References a settlement node that includes the transaction order to prove that it has been confirmed.

Correctness is jointly enforced by these script redeemers:

- Conclude of the transaction order spending validator (Section 2.3.2).
- Burn Transaction Order NFT of the transaction order minting policy (Section 2.3.1).
- Mint or Burn of the witness staking script (Section 2.5.1).

B.7 Reserve management

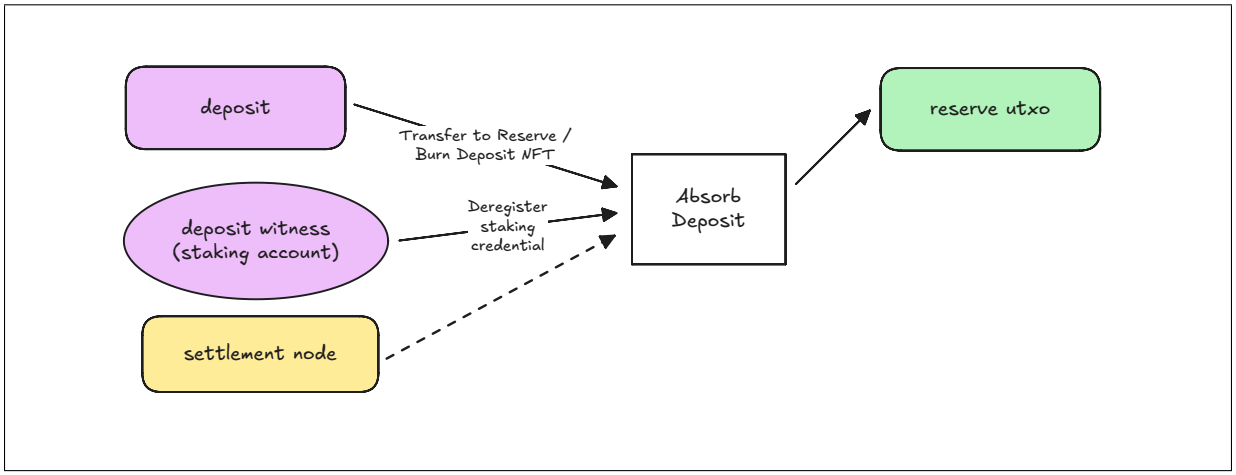

Figure B.7.1: Absorb a deposit into the Midgard reserves after it has been confirmed.

Absorb a confirmed deposit into Midgard reserves via a transaction that (Figure B.7.1):

1. Spends a deposit and burns its deposit NFT.
2. De-registers the staking credential witnessing the deposit.
3. References a settlement node that includes the deposit to prove that it has been confirmed.

Correctness is jointly enforced by these script redeemers:

- Transfer to Reserve of the deposit spending validator (Section 2.1.2).
- Burn Deposit NFt of the deposit minting policy (Section 2.1.1).
- Mint or Burn of the witness staking script (Section 2.5.1).

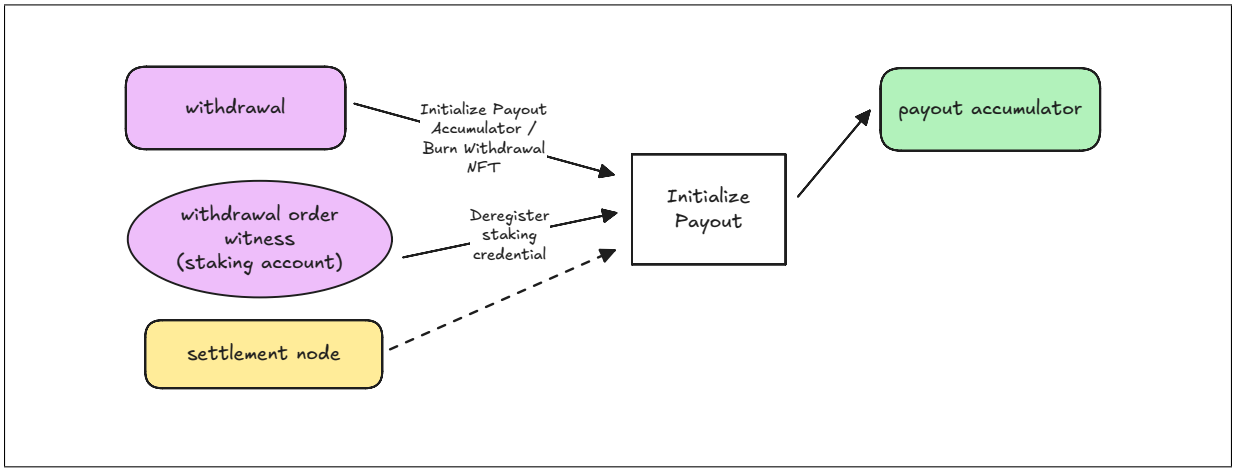

Figure B.7.2: Transform a confirmed withdrawal order into a payout accumulator.

Transform a confirmed withdrawal into a payout accumulator via a transaction that (Figure B.7.2):

1. Spends the withdrawal order and burns its NFT.
2. Produces a payout accumulator with `l2_value`, `l1_address`, and `l1_datum` datum fields matching those of the withdrawal order.
3. De-registers the staking credential witnessing the withdrawal order.
4. References a settlement node that includes the withdrawal to prove that it has been confirmed.

Correctness is jointly enforced by these script redeemers:

- Initialize Payout Accumulator of the withdrawal order spending validator (Section 2.4.2).
- Burn Withdrawal NFT of the withdrawal order minting policy (Section 2.4.1).
- Burn or Mint of the witness staking script (Section 2.5.1).

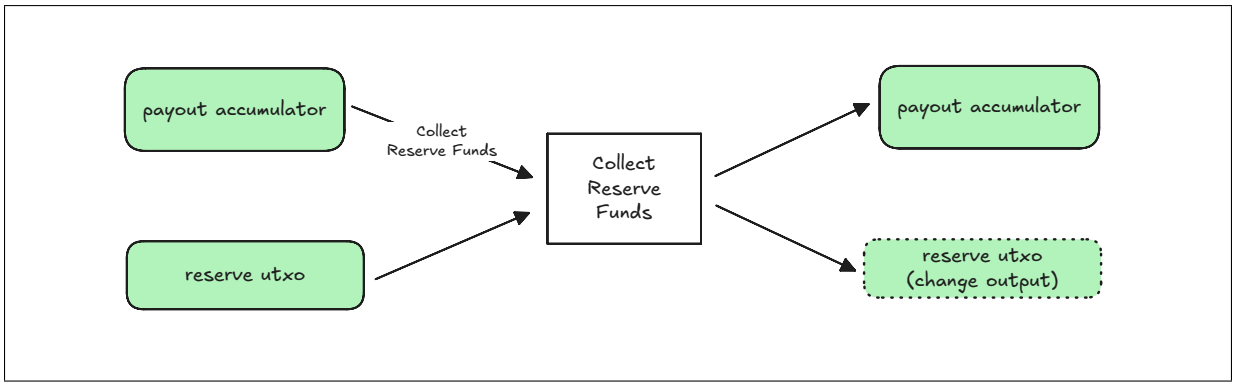

Figure B.7.3: Collect funds from Midgard reserves into a payout accumulator.

Collect funds from Midgard reserves into a payout accumulator via a transaction that (Figure B.7.3):

1. Spends a payout accumulator.
2. Spends a reserve utxo.
3. Produces a payout accumulator output that combines the funds from the payout accumulator input with as many funds from the reserve utxo input as possible, up to the limit defined in the payout accumulator's `l2_value` datum field.
4. Produces a reserve utxo if needed to hold the remainder of the funds.

Correctness is jointly enforced by these script redeemers:

- Collect Reserve Funds of the payout accumulator spending validator (**??**)
- The sole redeemer of the reserve spending validator (Section 3.7.1).

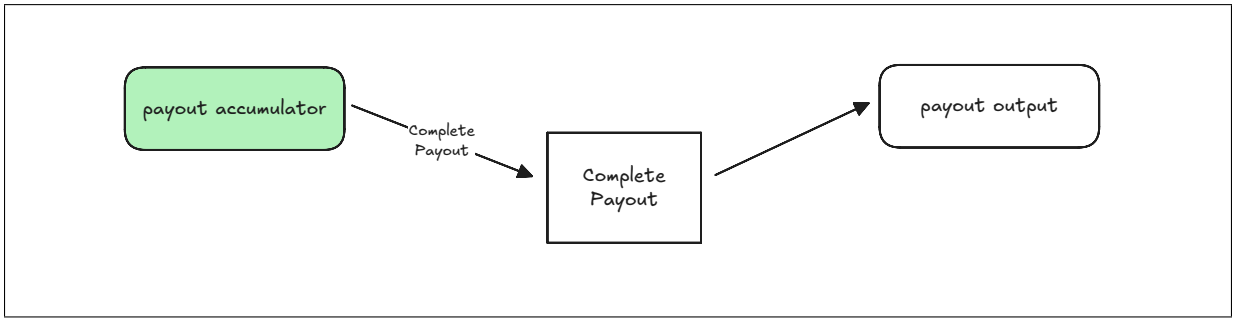

Figure B.7.4: Conclude a payout from Midgard.

Conclude a payout from Midgard via a transaction that (Figure B.7.4):

1. Spends a payout accumulator with funds matching or exceeding the amount defined in its `l2_value` datum field.
2. Produces an output at the address and datum defined by the payout accumulator's `l1_address` and `l1_datum` datum fields.

Correctness is jointly enforced by these script redeemers:

- Complete Payout of the payout accumulator spending validator (**??**).
- Burn of the payout accumulator minting policy (**??**).

B.8 Fraud proofs

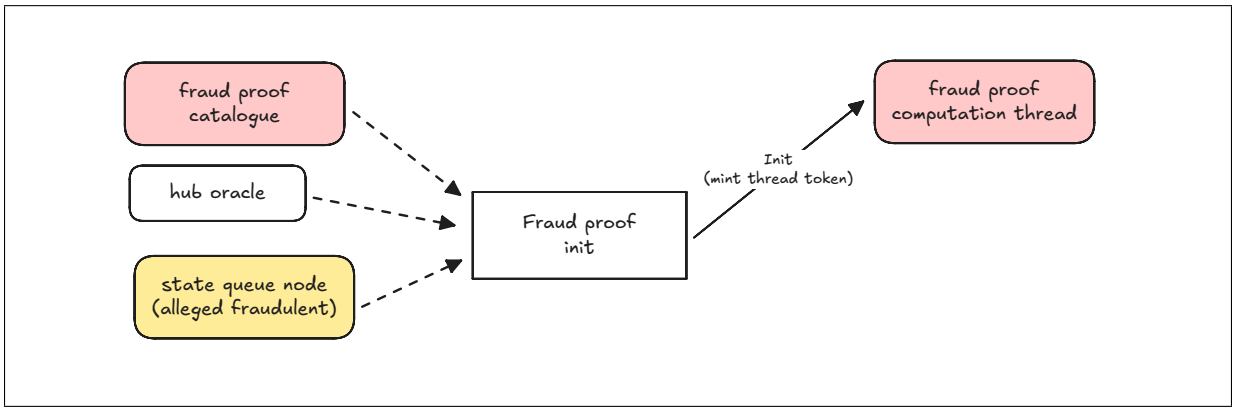

Figure B.8.1: Initialize the verification of a fraud proof.

Initialize the verification of a fraud proof via a transaction that (Figure B.8.1):

1. References an allegedly fraudulent node in the state queue.
2. References the fraud proof catalogue to select the fraud proof verification procedure that will demonstrate the allegation.
3. Mints a computation thread token and sends it to the first step spending validator of the selected fraud proof verification procedure.

Correctness is enforced by the Init redeemer of the fraud proof computation thread minting policy (Section 4.3.1).

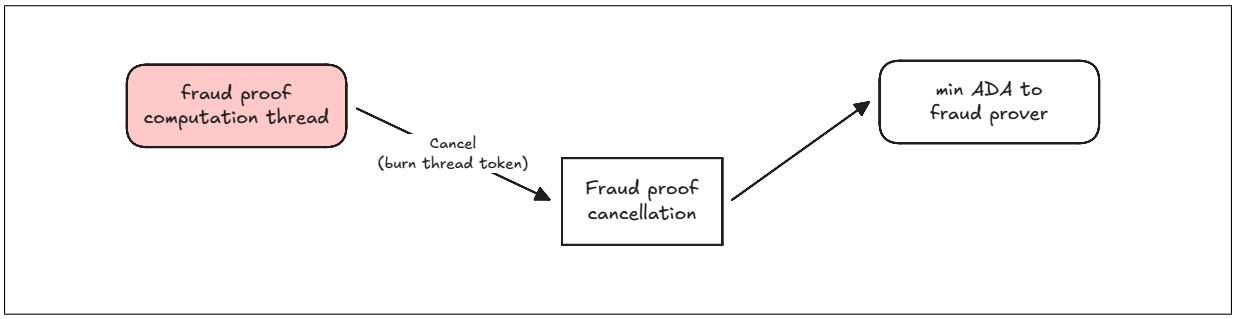

Figure B.8.2: Cancel the verification of a fraud proof.

Cancel the verification of a fraud proof via a transaction that (Figure B.8.2):

1. Spends a single fraud proof computation thread utxo.
2. Burns the computation thread token
3. Refunds the ADA from the computation thread utxo to the fraud prover.

Correctness is enforced by the Cancel redeemer of the fraud proof minting policy (Section 4.3.1).

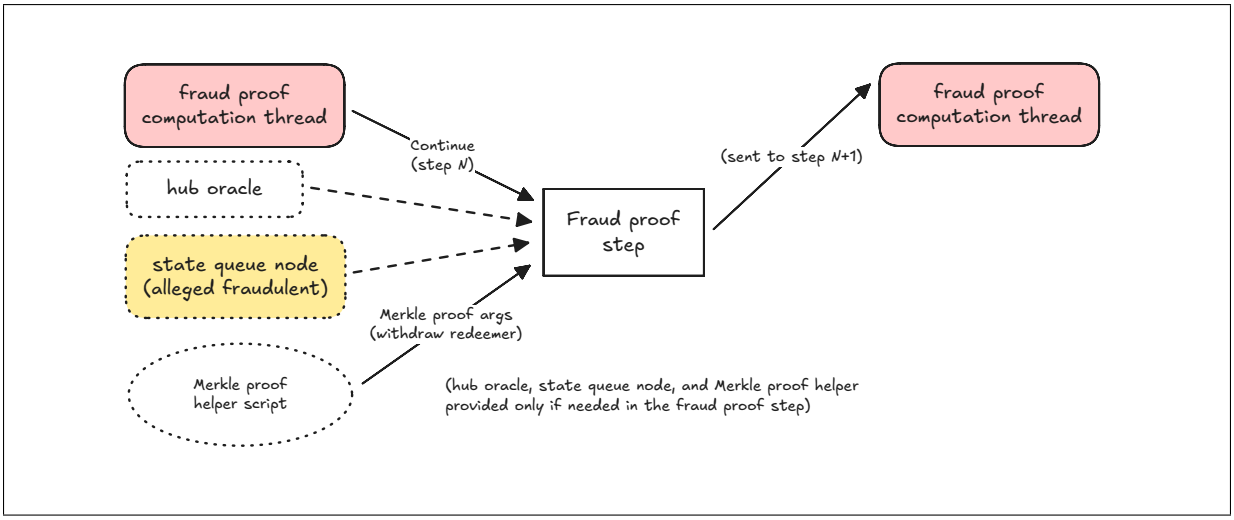

Figure B.8.3: Perform a verification step of a fraud proof.

Perform a single verification step of a fraud proof via a transaction that (Figure B.8.3):

1. Spends a single fraud proof computation thread utxo, performing the verification step defined in the utxo's spending validator.
2. Reproduces the fraud proof computation thread utxo as a continuing output sent to the next step's spending validator address.
3. If required by the verification step, references the allegedly fraudulent state queue node.
4. If required by the verification step, withdraws ADA from a designated account that triggers Midgard's Merkle proof helper script.

Correctness is jointly enforced by these script redeemers:

- Continue of the fraud proof's spending validator (Section 4.3.2).
- (If needed) Merkle Inclusion/Exclusion Proof of the helper script staking validator (see the Aiken Merkle Patricia Forestry repository on Github).

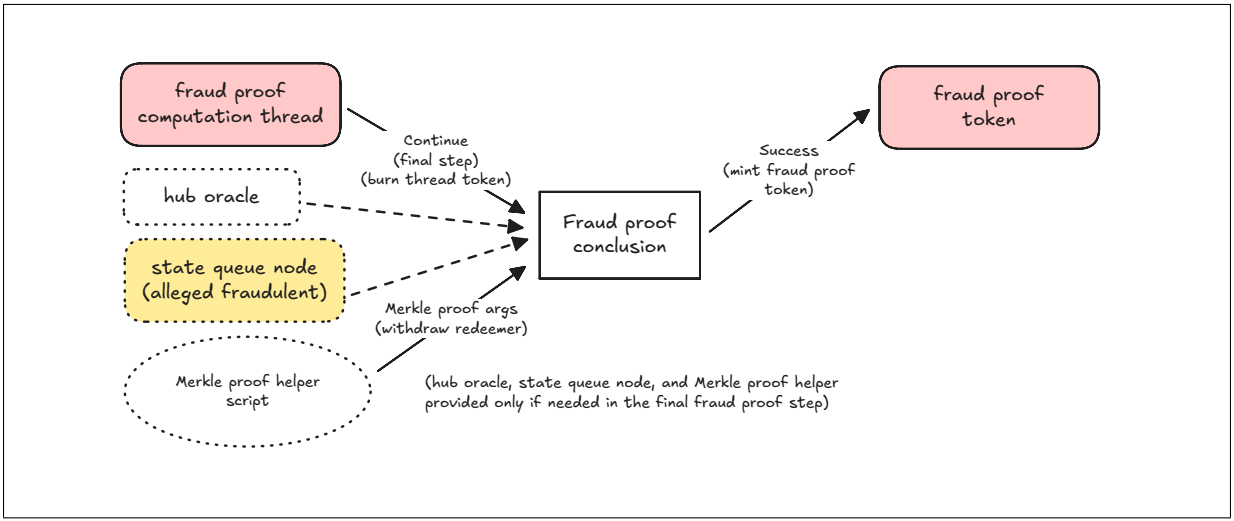

Figure B.8.4: Conclude the verification of a fraud proof by performing its final step.

Perform the final verification step of a fraud proof via a transaction that (Figure B.8.3):

1. Spends a single fraud proof computation thread utxo, performing the verification step defined in the utxo's spending validator.
2. Burns the computation thread token and mints the corresponding fraud proof token.
3. If required by the verification step, references the allegedly fraudulent state queue node.
4. If required by the verification step, withdraws ADA from a designated account that triggers Midgard's Merkle proof helper script.

Correctness is jointly enforced by these script redeemers:

- Mint of the fraud proof minting policy (Section 4.2.1).
- Conclude of the fraud proof's spending validator (Section 4.3.2).
- (If needed) Merkle Inclusion/Exclusion Proof of the helper script staking validator (see the Aiken Merkle Patricia Forestry repository on Github).

Appendix C

Design and implementation considerations

This appendix chapter describes some considerations that we have taken into account while designing and implementing the Midgard protocol.

C.1 Protocol invariants

Midgard's design must ensure that the following protocol invariants always hold. Auditors of the Midgard protocol are strongly encouraged to consider these invariants in their analysis.

Block merging order. Each valid block is merged to the confirmed state before its descendants.

If this invariant fails to hold, a block previously considered valid may become invalid because the confirmed state may unexpectedly change due to an out-of-order block being merged. This would significantly worsen the waiting time for L2 transaction confirmation, as users would not be able to rely on a block's validity at commitment time to hold during its maturity period.

Midgard enforces this invariant by using the linked list data structure in the state queue, which ensures that blocks are appended to the end of the queue and merged from the beginning of the queue. See Section 3.4.

Valid blocks. Every valid block can eventually be merged to the confirmed state.

If this invariant fails to hold, Midgard's confirmed state may become stuck because some valid block cannot be merged and cannot be removed from the state queue for fraud. This would be catastrophic because there would be no way to retrieve any funds from Midgard other than withdrawals that have already been confirmed by previous blocks but not yet paid out.

Midgard enforces this invariant by ensuring that the Merge to Confirmed State state queue redeemer always works for blocks that are mature with no unmerged predecessors left in the state queue. Furthermore, Midgard ensures that onchain verification scripts for fraud proofs can never succeed when targeting valid blocks. See Section 3.4 and Section 4.1.

Branch: (None) @ 8799ca9 • Commit date: 2025-10-22

Invalid blocks. Every invalid block can be invalidated by an L1-verified fraud proof before its maturity period elapses, allowing it to be removed from the state queue.

Suppose this invariant fails to hold for any violation type. In that case, dishonest operators can exploit the violation with impunity, and no one can prevent the invalid state from being merged to the confirmed state. Depending on the violation type, dishonest operators may use various exploits to steal user funds.

Midgard enforces this invariant by defining an onchain verification script for every possible violation of Midgard's ledger rules. Furthermore, Midgard ensures that these scripts can always succeed when targeting invalid blocks containing their corresponding violation type. See Section 4.1, Section 4.3, and Chapter 5.

Registered and active operators. Anyone with a sufficient ADA bond can eventually become an active operator, and every active operator will eventually get a shift to commit blocks.

If this invariant fails to hold, Midgard may fail to attract enough active operators to ensure robust and continuous block production. If all active operators cease participating in the Midgard protocol and no newcomers can become active operators or obtain a shift, then Midgard block production would cease.

Midgard enforces this invariant by using the linked list data structures to assign shifts to all active operators in a round-robin fashion. Furthermore, it uses another linked list to register new operators and ensures that they can always activate after a prescribed wait duration. See Section 3.2.3, Section 3.2.4, and Section 3.3.

Operator bond. No operator can retrieve their bond before the maturity period elapses for their most recent block, and every operator can do so afterward.

If the first clause of this invariant fails to hold, then a dishonest operator can prevent themself from being slashed and the fraud prover from being rewarded when a fraud proof removes the operator's fraudulent block from the state queue. This would weaken Midgard's incentives by reducing risk for dishonest operators and eliminating the reward for honest watchers, so that only altruistic watchers and those directly hurt by the fraud would be incentivized to stop it.

If the second clause of this invariant fails to hold, then people would hesitate to become operators due to the risk that their bond might get stuck in Midgard.

Midgard enforces this invariant through the operator directory, which tracks the maturity period of the latest block committed by each operator. It holds the operator bonds during these periods and allows them to be retrieved afterward. See Section 3.2.

Censorship protection. If block production continues, operators cannot censor deposits, withdrawal orders, and transaction orders from being confirmed.

If this invariant fails to hold, then some user funds may get stranded in Midgard because operators would prevent them from interacting with Midgard's state.

Midgard enforces this by assigning an inclusion time to every L1 user event (i.e.deposits, withdrawal orders, and transaction orders) and assigning an event interval to every block. Any block can be removed as invalid if it does not include an L1 user event with an inclusion time that falls within the block's event interval. The event intervals are non-overlapping

between blocks and have no gaps. This means that every L1 user event must eventually be included in a valid block unless block production halts.

Meanwhile, the ”registered and active operators” invariant and the escape hatch mechanism (see Section 3.5) mitigate the possibility that block production stops indefinitely. Thus, Midgard user events are protected from censorship.

C.2 Protocol parameters

C.2.1 Midgard consensus protocol parameters

The following parameters must be set before Midgard is initialized. Their actual values will be determined once Midgard is fully implemented, tested, and benchmarks. However, for the reader's intuition, we provide approximate values that we expect for the parameters:

Parameter	Definition	Expected approx. value
`event_wait_duration`	Section 2.1	2–4 minutes
`fraud_prover_reward`	Section 3.2	30%–50% of the `required_bond` parameter
`maturity_duration`	Section 3.4	3–7 days
`registration_duration`	Section 3.2	1 day
`required_bond`	Section 3.2	50K–200K ADA
`shift_duration`	Section 3.1	1 hour
`slashing_penalty`	Section 3.2	50%–70% of the `required_bond` parameter

Table C.1: Midgard consensus protocol parameters.

Midgard does not use the Ouroboros consensus protocol, so it does not need to set the associated protocol parameters.

C.2.2 Midgard ledger parameters

Midgard's L2 transactions transition its ledger similarly to Cardano's ledger but with the exceptions defined in Section 1.6.2. Consequently, Midgard uses the same ledger-associated protocol parameters as Cardano but omits certain parameters as follows:

- No staking or governance parameters.
- No parameters for obsolete pre-Conway features.

C.2.3 Midgard fee structure

Midgard will collect fees from all L2 transactions, in a similar way to how fees are collected for Cardano L1 transactions. Furthermore, Midgard may collect fees for processing deposits and withdrawals, as these user events incur costs separate from L2 transactions. However, Midgard's fee parameters are expected to be orders of magnitude smaller than Cardano's fee parameters.

Midgard's L1 operating costs per block are fixed, regardless of the number and complexity of transactions in the block. Midgard's DA temporary storage costs per block are proportional to the number and size of transactions in the block, but these variable costs are orders of magnitude smaller than Cardano L1 storage costs. Midgard's revenue per block is proportional to the number and complexity of transactions in the block. Therefore, on a per-block basis, Midgard's fee revenue grows faster than its costs as the number of transactions in the block increases. This means that Midgard can sustain much lower L2 transaction fees once it attains a certain average level of L2 activity.

The specific values of Midgard's fee parameters will be determined before launch based on simulations, benchmarks, and community feedback.

C.2.4 Midgard network parameters

Technically, Midgard operator nodes do not need to communicate with each for consensus. Instead, they participate in the consensus protocol by interacting with Midgard's smart contracts on L1.

However, in practice, an offchain peer-to-peer gossip network between operators may help them run things a bit more smoothly for users. If so, there may be associated protocol parameters to help peers discover the network topology and communicate with other peers.

C.3 Protocol security

C.3.1 Malicious majority stake attack

A majority stake attack can allow the adversary to degrade the Liveness and Finality security properties of a blockchain protocol, if the adversary holds a sufficient percentage (e.g. 51%) of the network's primary resource (e.g. hashing power for Bitcoin, ADA for Cardano). In practical terms, this attack can allow the adversary to revert recently confirmed transactions and extract value from victims by preventing the reverted transactions from being restored to the ledger.

Midgard's consensus protocol is implemented entirely via Cardano L1 smart contracts. This means that an attack to revert any transaction that evolves the state of these L1 smart contract is as difficult as an attack against Cardano's finality security property, against which Cardano's Ouroboros consensus protocol is highly resistant. In simpler terms, Cardano's full ADA supply on L1 is the resource against which a majority stake attack must prevail to affect Midgard's L2 ledger.

Furthermore, every L2 transaction in Midgard is only confirmed after two consensus transactions occur on L1:

- First, the L2 transaction is included in a block header committed to Midgard's state queue.
- Second, the block header is merged to Midgard's confirmed state after waiting in the queue for the maturity period.

Both of the above L1 transactions must be reverted to revert a confirmed L2 transaction in Midgard. Due to the mandatory maturity period between the two L1 transactions, this amounts to a long-range attack to fork Cardano's blockchain before the maturity period began. However, according to the Ouroboros consensus protocol, Cardano nodes will always reject any forks that diverge from their local chain by more than 2160 blocks, which is approximately 12 hours (with very tight confidence bounds). As Midgard's maturity period protocol parameter will certainly be longer than 24 hours (see Section C.2), it will be impossible to revert both consensus transactions on L1 for a confirmed L2 transaction on Midgard.

Suppose the adversary succeeds in an attack against Cardano's finality property, reverting the second consensus transaction for a Midgard L2 transaction. This reversion will restore the block header to the beginning of the state queue, allowing it to be immediately re-merged to Midgard's confirmed state. Moreover, no conflicting block header can be merged to the confirmed state in its place. Thus, the adversary's attack has no practical effect other than slightly delaying the inevitable confirmation.

Suppose the adversary reverts the first consensus transaction for a Midgard L2 transaction. This reversion will remove the block header from the state queue. If the adversary is *not* colluding with the current operator in Midgard, then the current operator will simply recommit the block header to the state queue, nullifying any effects of the adversary's attack. If the adversary is colluding with the current operator, then the adversary's attack is redundant—the same effect is achieved if the current operator abuses its power to censor L2 transactions. However, Midgard has safeguards in place against such abuse (see Section C.3.2).

C.3.2 Operator abuse of power

Each operator has the exclusive privilege to commit blocks to the state queue and resolve nodes in the settlement queue during their assigned shifts (see Section 3.1), the duration of which is controlled by the `shift_duration` protocol parameter (see Section C.2). The current operator has discretion over whether and when to commit blocks to the state queue and the contents of those blocks. This discretion can be abused to censor L2 transactions, but Midgard's consensus protocol has safeguards in place against such abuse.

Midgard deposits and withdrawals are initiated via L1 smart contracts that assign definite inclusion times to them. An operator block is invalid if it contains these inclusion times in its event interval but fails to include the associated deposit or withdrawal events. This ensures that if operators continue committing blocks to Midgard's state queue, then they cannot ignore deposit and withdrawal events.

Midgard L2 transaction requests are typically submitted to operators via a publicly accessible API, and they can be ignored by operators. However, any user can escalate his L2 transaction request by posting a transaction order on L1. Similar to Midgard deposits and withdrawals, an L1 transaction order is assigned an inclusion time that guarantees its inclusion in a subsequent valid block.

If Midgard operators stop committing blocks at all to the state queue, then the inclusion times on their own cannot guarantee that deposits, withdrawals, and L2 transactions will be processed in a timely manner. However, for this extreme case, Midgard's consensus protocol includes the escape hatch mechanism, which temporarily lowers the bond requirement for new operators. This makes it easier for honest actors to become operators and restart block production. After the emergency is over, the new operators must either retire or increase their ADA bond to the normal requirement to stay as regular operators.

C.3.3 Replay attack

In a replay attack, the adversary repeats or delays a valid message to confuse the network and achieve a malicious outcome.

Midgard's entire consensus protocol is implemented as a set of smart contracts on Cardano L1 that evolve via L1 transactions. Each of these L1 transactions must spend at least one utxo, which means that Cardano's ledger rule against double-spending utxos prevent these transactions from being replayed.

However, Cardano's ledger rules do not directly see the contents corresponding to any hashes in the consensus protocol state on L1, such as the utxos, L2 transactions, deposits, and withdrawals in a block header. Nonetheless, Midgard has comprehensive ledger rules against the various ways in which utxos, L2 transactions, deposits, and withdrawals can be duplicated within a block or across blocks. Any operator that commits a block header to the state queue that contravenes these ledger rules forfeits his bond if the corresponding fraud proof is verified on L1 within the block's maturity period.

Another form of replay attack involves reusing the same confirmed deposit or withdrawal event in the settlement queue to absorb more deposited funds than necessary into Midgard's reserve or release more funds than necessary from it for a withdrawal. Midgard prevents this by requiring an L1 utxo to be created on L1 for every deposit and withdrawal, which must be spent whenever the corresponding deposit is absorbed or withdrawal is paid out.

List of Figures